Universitatea din București - Facultatea de Matematică și Informatică

Evoluția structurilor de date
Structuri de date dinamice

Student: Oprea Tudor
Grupa: 241

iunie 2023
București

Cuprins:

1. Ce sunt structurile de date?

Structurile de date sunt modalități de organizare și stocare a datelor într-un format eficient, astfel încât să poată fi accesate, manipulate și gestionate într-un mod optim. Ele reprezintă un aspect fundamental al informaticii și sunt utilizate în toate tipurile de aplicații software.

O structură de date definește modul în care datele sunt organizate în memoria unui sistem informatic, precum și operațiile care pot fi efectuate asupra acestor date. Aceasta include modul în care datele sunt stocate într-un anumit format și cum sunt interconectate pentru a permite accesul și manipularea eficientă a acestora.

2. Clasificare

Structurile de date pot fi clasificate în funcție de modul în care sunt organizate și modul în care permit accesul și manipularea datelor. Există mai multe criterii de clasificare a structurilor de date, iar în continuare vom explora cele mai comune categorii.

1. Structuri de date liniare:

Acestea sunt structuri în care elementele sunt organizate într-o secvență liniară. Accesul la elemente se face într-un mod secvențial, de la un capăt la celălalt. Exemple includ matricea (array), lista liniară (linked list), coada (queue) și stiva (stack).

2. Structuri de date ierarhice:

Acestea sunt structuri care permit organizarea într-o ierarhie, în care fiecare element poate avea un părinte și zero sau mai mulți copii. Exemple includ arborii (trees), cum ar fi arborii binari și arborii de căutare echilibrată.

3. Structuri de date grafice:

Acestea sunt structuri care permit reprezentarea relațiilor complexe între elemente prin intermediul nodurilor și muchiilor. Grafurile pot fi orientate sau neorientate și pot avea diverse proprietăți. Aceste structuri sunt utilizate pentru a modela rețele, dependențe, relații sociale etc.

4. Structuri de date tabulare:

Acestea sunt structuri care organizează datele în tabele sau matrice bidimensionale. Un exemplu cunoscut este tabela hash (hash table), care utilizează o funcție de hash pentru a accesa rapid elementele pe baza cheilor.

5. Structuri de date speciale:

Această categorie include structuri de date care sunt concepute pentru a rezolva probleme specifice sau pentru a optimiza anumite operații. Exemple includ heap-ul binar, arborele de sintaxă, trie-ul și multe altele.

În plus, structurile de date pot fi clasificate și în funcție de modul în care permit accesul la date și modul în care acestea sunt actualizate. De exemplu, structurile de date pot fi statice

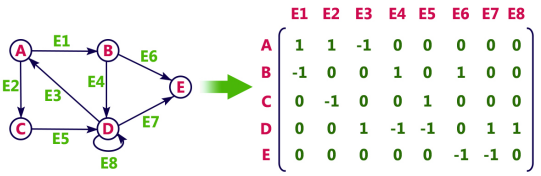
(unde dimensiunea este predefinită și nu poate fi modificată) sau dinamice (unde dimensiunea poate crește sau scădea pe măsură ce datele sunt adăugate sau eliminate). Clasificarea structurilor de date este importantă pentru alegerea celei mai potrivite pentru o anumită problemă sau aplicație. În funcție de cerințe și operațiile pe care dorim să le efectuăm asupra datelor, putem alege o structură de date optimă care să ofere eficiență și performanță

3. Începuturile

Primele structuri de date au apărut odată cu apariția informaticii ca disciplină științifică și practică în anii '50 și '60. În această perioadă, s-au pus bazele teoretice și practice ale programării și organizării datelor în calculatoare.
Iată câteva exemple de prime structuri de date și aspectele asociate acestora:

Matrice (Array):

Matricea este una dintre primele structuri de date utilizate în informatică. Aceasta constă într-o colecție liniară de elemente de același tip, stocate într-o secvență continuă de memorie. Matricea poate avea o sau mai multe dimensiuni, iar accesul la elemente se realizează prin intermediul unui index numeric.
Matricele erau utilizate pentru a stoca și accesa seturi de date structurate, cum ar fi tabele sau imagini. Acestea au oferit o modalitate simplă de organizare a datelor într-un format tabular.



Matricea este una dintre cele mai simple și ubiquitare structuri de date în informatică. Ea reprezintă un tablou uni/bidimensional de elemente, organizate în rânduri și coloane. Fiecare element poate fi accesat prin intermediul unui set de indici, care reprezintă poziția sa în matrice. Matricele sunt utilizate într-o gamă largă de aplicații, inclusiv în grafică computerizată, prelucrarea imaginilor, știința datelor și altele. Acestea oferă o modalitate eficientă de a stoca și accesa date într-o structură organizată, permitând manipularea lor în funcție de necesități specifice. Operațiile comune pe matrice includ inserția și ștergerea elementelor, căutarea, sortarea și manipularea datelor pe rânduri și coloane. Prin utilizarea matricelor și algoritmilor corespunzători, se poate realiza o serie de operații complexe și analize pe date structurate, contribuind astfel la dezvoltarea soluțiilor software puternice și eficiente.

Lista înlănțuită (Linked List):

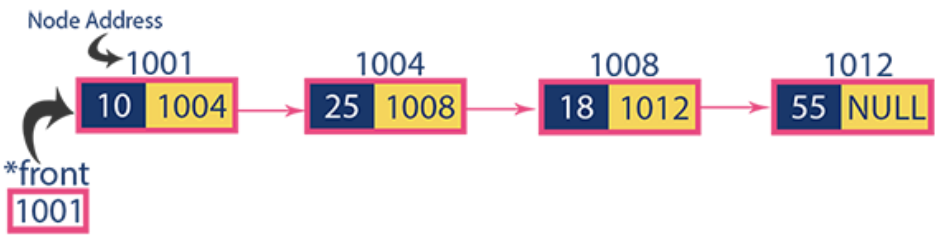
Lista înlănțuită este o structură de date în care elementele sunt stocate în noduri legate între ele. Fiecare nod conține o valoare și o referință către următorul nod din listă.
Aceasta a permis o flexibilitate mai mare în gestionarea datelor decât matricea.
Listele înlănțuite sunt utilizate pentru a stoca și manipula date într-o ordine arbitrară. Ele au fost deosebit de utile în cazul în care dimensiunea sau ordinea elementelor putea varia în timpul execuției programului.

Principalele avantaje ale listei înlănțuite sunt capacitatea de a permite inserții și ștergeri eficiente în orice poziție a listei, fără a necesita realocarea memoriei. Aceasta face ca lista înlănțuită să fie utilă în situații în care dimensiunea și structura datelor se modifică frecvent.

Există două tipuri principale de liste înlănțuite:

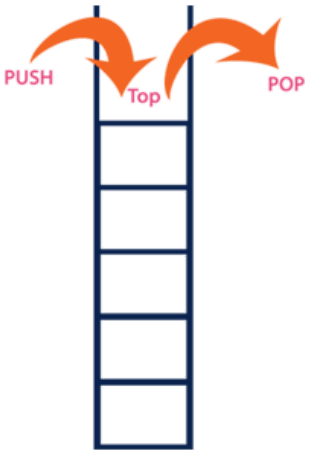
1. Listă înlănțuită simplă (singly linked list): În acest tip de listă, fiecare nod conține o valoare și o referință către următorul nod din listă. Ultimul nod are o referință către "NULL". Accesul la elemente într-o listă înlănțuită simplă se face într-o manieră secvențială, începând de la primul nod și continuând până la găsirea elementului dorit sau până la sfârșitul listei.
2. Listă înlănțuită dublă (doubly linked list): În acest tip de listă, fiecare nod conține o valoare, o referință către nodul anterior și o referință către nodul următor din listă. Această structură permite accesul bidirecțional la elementele listei și facilitează operații precum inversarea listei sau parcurgerea de la sfârșit către început.

Lista înlănțuită oferă o flexibilitate mare în gestionarea datelor și permite implementarea eficientă a operațiilor de inserție și ștergere în orice poziție a listei. Cu toate acestea, accesul la elemente într-o listă înlănțuită necesită parcurgerea secvențială a listei, ceea ce poate avea un impact asupra performanței în cazul accesului frecvent sau căutării în listă.
În concluzie, lista înlănțuită este o structură de date versatilă, utilizată într-o varietate de aplicații, care oferă flexibilitate și eficiență în gestionarea și organizarea datelor într-o succesiune liniară.



Stiva (Stack) și Coadă (Queue):

Stiva și coada sunt două structuri de date liniare care au apărut în aceeași perioadă. Stiva funcționează pe principiul "Last In, First Out" (LIFO), în timp ce coada funcționează pe principiul "First In, First Out" (FIFO).

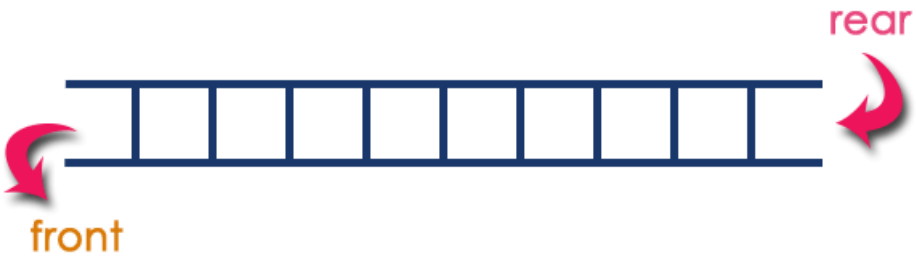


Stiva a fost utilizată pentru gestionarea apelurilor de funcții într-un program, stocând adresele de revenire și variabilele locale. Operațiile principale pe o stivă sunt:

- Push: Adaugă un element la vârful stivei.
- Pop: Extrage și elimină elementul de la vârful stivei.
- Top: Returnează valoarea elementului de la vârful stivei, fără a-l extrage.

Stiva este utilizată într-o gamă largă de aplicații, cum ar fi evaluarea expresiilor matematice, implementarea algoritmilor de backtracking și gestionarea apelurilor de funcții într-un limbaj de programare.

Coada a fost utilizată pentru gestionarea evenimentelor într-un sistem, cum ar fi procesarea mesajelor primite.



Funcționează pe principiul unei linii de așteptare, unde elementele noi sunt adăugate la sfârșitul cozii și elementele sunt eliminate din capătul opus. Operațiile principale pe o coadă sunt:

- Enqueue: Adaugă un element la sfârșitul cozii.
- Dequeue: Extrage și elimină elementul din capătul opus al cozii.
- Front: Returnează valoarea elementului din capătul opus al cozii, fără a-l extrage.

Coada este utilizată într-o varietate de aplicații, precum planificarea proceselor în sistemele de operare, gestionarea cererilor într-un server sau implementarea algoritmilor de parcurgere a grafurilor.

Atât stiva, cât și coada sunt structuri de date eficiente pentru stocarea și manipularea datelor într-o ordine specifică. Ele oferă operații simple și rapide pentru adăugarea și extragerea elementelor, permitând implementarea eficientă a algoritmilor și rezolvarea problemelor complexe în informatică.

Aceste prime structuri de date au oferit bazele pentru organizarea și manipularea datelor într-un mod structurat și eficient. Deși ele erau relativ simple, aceste concepte fundamentale au pus bazele pentru dezvoltarea și optimizarea ulterioară a structurilor de date mai complexe și eficiente.
Este important de menționat că evoluția structurilor de date a continuat în decursul anilor, iar cercetătorii și dezvoltatorii au creat și optimizat o varietate de alte structuri pentru a satisface nevoile tot mai complexe ale aplicațiilor software.

Este important de menționat și faptul că primele structuri de date au evoluat de-a lungul timpului, fiind rezultatul muncii și contribuțiilor multiple ale cercetătorilor și pionierilor din domeniul informaticii. Nu există o persoană unică care să fi inventat toate aceste structuri de date, deoarece ele au fost dezvoltate și rafinate colectiv în comunitatea informatică.

Deși nu putem atribui în mod specific inventarea structurilor de date individuale unei singure persoane, există câțiva cercetători și personalități remarcabile care au contribuit semnificativ la dezvoltarea acestor concepte. Iată câțiva dintre ei:

- Grace Hopper: A fost o importantă pionieră în domeniul informaticii și una dintre primele persoane care a propus utilizarea termenului "listă liniară" (linked list) și "coadă" (queue) pentru structurile de date asociate.
- John von Neumann: Este cunoscut pentru arhitectura von Neumann, care a reprezentat un punct de referință pentru dezvoltarea calculatoarelor moderne. Conceptele sale au fost fundamentale pentru organizarea și manipularea datelor în memorie, inclusiv utilizarea matricelor (array).
- Donald Knuth: Este unul dintre cei mai influenți matematicieni și informaticieni din istoria informaticii. Lucrările sale, cum ar fi seria "The Art of Computer Programming", au explorat și au oferit detalii teoretice și practice despre diverse structuri de date și algoritmi.
- Edsger Dijkstra: A fost unul dintre cei mai importanți cercetători în domeniul științei calculatoarelor și a contribuit la dezvoltarea conceptelor de stivă (stack) și coadă (queue) și a algoritmilor asociati, precum și a algoritmilor de căutare și sortare.

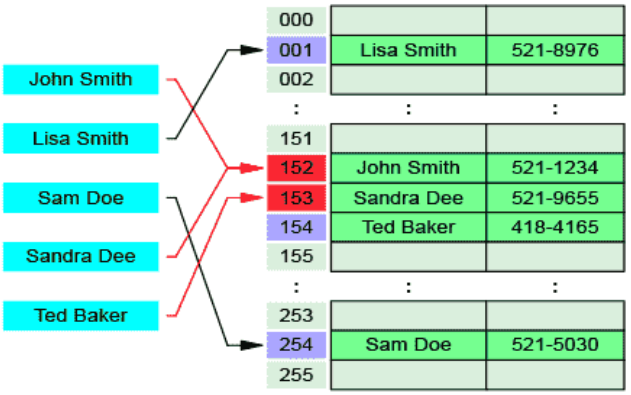
Acești pionieri și mulți alții au contribuit la dezvoltarea și rafinarea structurilor de date, oferind fundația teoretică și practică necesară pentru evoluția informaticii. Este important să recunoaștem că structurile de date sunt rezultatul unei colaborări și evoluții continue în comunitatea informatică, iar atribuirea exactă a inventării poate fi dificilă.

4. Evoluția ulterioară

După primele structuri de date, dezvoltarea și evoluția continuă a adus o serie de structuri mai noi și mai complexe, adaptate pentru a satisface cerințele tot mai diverse ale aplicațiilor software. Iată câteva exemple de structuri de date mai noi:

Hash Table (Tabel de dispersie):

Tabela hash este o structură de date care utilizează o funcție de hash pentru a asocia chei cu valori. Ea oferă o eficiență excelentă în găsirea și inserarea datelor, având o complexitate constantă în cazul cel mai bun. Tabelele de dispersie sunt utilizate într-o varietate de aplicații, inclusiv baze de date, cache-uri și algoritmi de căutare eficientă.



Principalele caracteristici ale unei tabele hash sunt:

1. Funcția de dispersie: Este o funcție care preia o cheie și returnează un indice în tabloul de bucăți. O funcție de dispersie ideală distribuie cheile în mod uniform în întregul tablou, minimizând coliziunile (situația în care două chei diferite au aceeași valoare hash).
2. Tablou de bucăți: Este un tablou de dimensiune fixă care stochează cheile și valorile asociate. Fiecare bucată poate conține una sau mai multe perechi cheie-valoare. În cazul unei coliziuni, adică atunci când două chei diferite sunt dispersate în aceeași bucată, ele pot fi stocate într-un mod specific (de exemplu, utilizând o listă înlănțuită sau o altă structură de date pentru a gestiona coliziunile).
3. Operații principale:
    - Inserare: Adaugă o pereche cheie-valoare în tabela de dispersie. Funcția de dispersie este aplicată cheii pentru a determina bucata în care să fie stocată.
    - Căutare: Găsește valoarea asociată unei chei date în tabela de dispersie. Funcția de dispersie este aplicată cheii pentru a determina bucata în care să se facă căutarea.

- Ștergere: Elimină o pereche cheie-valoare din tabela de dispersie, pe baza cheii specificate. Funcția de dispersie este aplicată cheii pentru a localiza bucata în care se găsește perechea de șters.

Avantajele utilizării unei tabele de dispersie sunt:

- Acces rapid la elemente: Datorită funcției de dispersie eficiente, căutarea, inserarea și ștergerea se pot face în timp constant (O(1)), în cel mai bun caz.
- Eficiență în gestionarea unui număr mare de date: Tabelele hash sunt eficiente în gestionarea volumelor mari de date, oferind performanță constantă în majoritatea operațiilor, indiferent de dimensiunea datelor.

Unul dintre principalele dezavantaje ale tabelelor de dispersie este utilizarea spațiului. Dimensiunea tabloului de bucăți trebuie să fie suficient de mare pentru a evita coliziunile, ceea ce poate duce la utilizarea excesivă a memoriei. De asemenea, o funcție de dispersie ineficientă poate provoca coliziuni frecvente, ceea ce afectează performanța tabelului de dispersie.

Grafuri ponderate:

Grafurile ponderate sunt o extensie a grafurilor clasice, în care muchiile sunt asociate cu o valoare (sau pondere). Acestea sunt utilizate în algoritmi de căutare a celor mai scurte drumuri, algoritmi de arbori de acoperire minimă și alte aplicații care implică evaluarea și compararea costurilor între noduri și muchii.

Caracteristicile cheie ale grafurilor ponderate sunt următoarele:

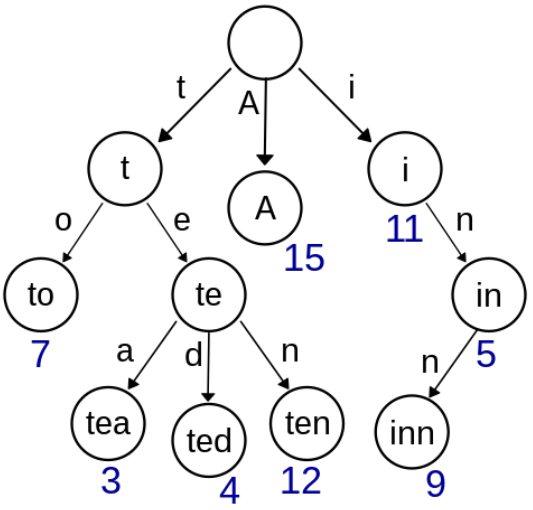
1. Muchii ponderate: Fiecare muchie din graf are o valoare numerică asociată, numită ponderare sau cost. Aceste ponderi pot fi pozitive, negative sau nule, în funcție de interpretarea specifică a grafului și a domeniului de aplicare.
2. Direcție și orientare: Grafurile ponderate pot fi orientate sau neorientate. În cazul grafurilor orientate, ponderile se aplică direcționat, adică pot exista ponderi diferite pe muchiile care conectează aceleași două noduri, dar în direcții opuse. În cazul grafurilor neorientate, ponderile sunt considerate identice pe ambele direcții ale muchiilor.
3. Algoritmi de căutare și explorare: Grafurile ponderate pot fi utilizate în diverse probleme, cum ar fi găsirea celui mai scurt drum între două noduri, găsirea arborelui de acoperire minim, planificarea rutelor în rețele de transport sau în alte aplicații care implică optimizarea costului sau eficienței.
4. Algoritmi de distanță minimă: Unul dintre principalele aspecte ale grafurilor ponderate este găsirea celui mai scurt drum între două noduri. Pentru aceasta, se utilizează algoritmi precum Dijkstra și Bellman-Ford, care iau în considerare ponderile muchiilor pentru a determina distanțele minime.
5. Greutăți și cicluri negative: Grafurile ponderate pot avea greutăți negative pe muchii, ceea ce poate duce la apariția ciclurilor negative în graf. În astfel de cazuri, determinarea celor mai scurte căi devine mai complexă, deoarece nu

există o soluție optimă. Algoritmul Bellman-Ford poate detecta și gestiona ciclurile negative, în timp ce algoritmul Dijkstra nu funcționează în prezența acestora.

Grafurile ponderate sunt utile într-o varietate de domenii, cum ar fi planificarea rutelor, navigarea GPS, rețele de comunicații, analiza rețelelor sociale, optimizarea resurselor și multe altele. Utilizarea ponderilor permite o analiză și o optimizare mai precisă a problemelor, luând în considerare costurile asociate cu conexiunile și distanțele între noduri.

Trie (Prefix Tree):

Trie este o structură de date utilizată în special pentru stocarea și căutarea eficientă a cuvintelor și prefixelor acestora. Este optimizat pentru a permite căutări rapide în dicționare, autocompletare și alte operații care implică manipularea și analiza textului.



Principalele caracteristici ale unui trie sunt:

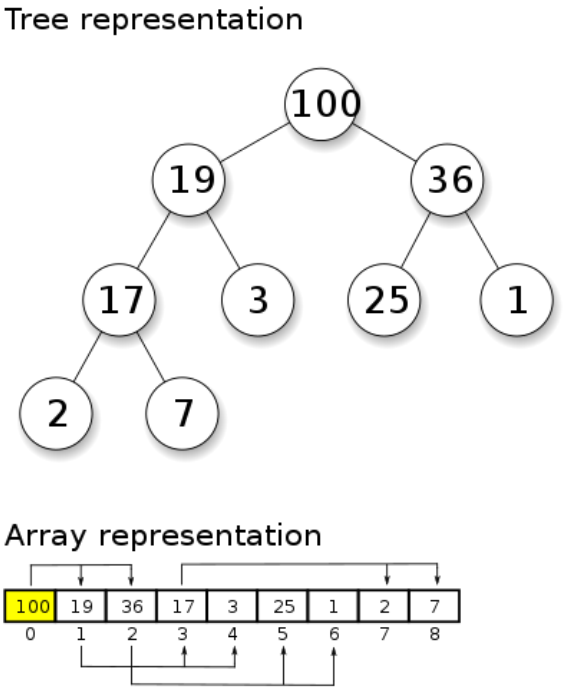
1. Structură ierarhică: Trie este o structură ierarhică, similară unui arbore, în care fiecare nod reprezintă o literă sau un caracter dintr-un șir de caractere. Rădăcina trie-ului reprezintă un nod vid sau un nod special fără valoare, iar fiecare nod intern poate avea zero sau mai mulți copii.
2. Noduri de sfârșit de cuvânt: Unele noduri din trie sunt marcate ca "noduri de sfârșit de cuvânt", indicând că un cuvânt se termină la acel nod. Aceasta permite trie-ului să distingă între cuvinte diferite și să realizeze căutări precise.
3. Eficiență în căutare: Trie oferă o căutare eficientă a șirurilor de caractere, deoarece numărul de comparații necesare este proporțional cu lungimea șirului, indiferent de dimensiunea totală a trie-ului. Astfel, timpul de căutare într-un trie este O(L), unde L reprezintă lungimea șirului de caractere căutat.

4. Gestionarea eficientă a memoriei: Trie utilizează memorie eficient, deoarece nodurile comune între cuvinte sunt partajate. Acest lucru reduce redundanța și economisește spațiu în comparație cu alte structuri de date, cum ar fi tabelele de dispersie.
5. Operații de inserare și ștergere: Trie permite inserarea și ștergerea eficientă a șirurilor de caractere. În timpul inserării, se construiește arborele pentru a reprezenta noul șir, iar în timpul ștergerii, se elimină nodurile corespunzătoare șirului, păstrându-se coerența trie-ului.

Trie este utilizat într-o varietate de aplicații care implică căutarea și stocarea șirurilor de caractere. Exemple includ autocompletarea în căutarea pe bază de text, verificarea rapidă a existenței unui cuvânt într-un dicționar, rezolvarea problemelor de prefix sau sufix și multe altele. Datorită eficienței sale în căutare și gestionarea eficientă a memoriei, trie este o alegere populară pentru scenariile în care se lucrează cu mulțimi mari de șiruri de caractere.

Heap:

Heap-ul este o structură de date de tip arbore binar, care are proprietăți speciale de ordonare. Este folosit în special pentru a obține acces rapid la cel mai mare sau cel mai mic element dintr-un set de date. Heap-ul este utilizat în algoritmi de sortare (de exemplu, Heap Sort) și în alte aplicații care necesită gestionarea rapidă a elementelor în ordine parțială.



Principalele caracteristici ale unui heap sunt:

1. Proprietatea de heap: Într-un heap, pentru fiecare nod, valoarea acestuia este mai mare sau egală cu valorile fiilor săi (în cazul unui heap maxim) sau mai mică sau egală (în cazul unui heap minim). Aceasta înseamnă că cel mai mare (sau cel mai mic) element se găsește întotdeauna la rădăcina heap-ului.
2. Arbore binar complet: Heap-ul este adesea implementat ca un arbore binar complet, ceea ce înseamnă că toate nivelurile, cu excepția posibil a celui mai de jos nivel, sunt complet umplute. Dacă există noduri în cel mai de jos nivel, acestea sunt așezate de la stânga la dreapta, fără goluri.
3. Operații de bază: Heap-ul oferă câteva operații de bază pentru gestionarea elementelor:
    - Inserare: Adaugă un element în heap în conformitate cu proprietatea de heap. Elementul este plasat într-o poziție adecvată pentru a păstra proprietatea de ordine parțială.
    - Extragerea maximului (sau minimului): Elimină și returnează cel mai mare (sau cel mai mic) element din heap. Heap-ul este reorganizat astfel încât să păstreze proprietatea de heap.
    - Accesul la maxim (sau minim): Returnează valoarea maximă (sau minimă) din heap fără a o elimina.
    - Modificarea unei valori: Permite modificarea unei valori existente în heap și reorganizarea acestuia pentru a menține proprietatea de heap.
4. Utilizare în sortare: Heap-urile pot fi utilizate și pentru a realiza sortarea elementelor. Aceasta implică construirea unui heap din setul de elemente și repetarea extragerii succesive a maximului (sau minimului) pentru a obține elementele sortate în ordine crescătoare (sau descrescătoare).

Heap-urile sunt utilizate într-o varietate de aplicații, cum ar fi algoritmi de sortare eficienți (de exemplu, Heap Sort), gestionarea priorităților, găsirea celor mai mari (sau mai mici) k elemente dintr-un set, planificarea de evenimente și multe altele. Datorită proprietăților sale de ordine parțială și eficienței operațiilor de bază, heap-ul oferă o soluție eficientă și versatilă pentru gestionarea și manipularea seturilor de date.

Acestea sunt doar câteva exemple de structuri de date mai noi care au apărut în urma cercetării și dezvoltării continue în domeniul informaticii. În plus, există multe alte structuri și variante ale structurilor menționate, precum și optimizări și extensii specifice pentru diverse aplicații și contexte. Evoluția continuă a structurilor de date reflectă nevoia constantă de a aborda probleme complexe și de a optimiza performanța în diferite domenii de aplicații.

5. Structuri de date randomizate

5.1. Skip Lists

Introduse in 1989 de către W. Purgh, sunt structuri dinamice bazate pe factor aleator (randomized)
”Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.”
— William Pugh, Concurrent Maintenance of Skip Lists (1989)

Skip list este o structură de date probabilistică, bazată pe liste înlănțuite, care permite accesul rapid și eficient la elemente într-o colecție ordonată de date. Skip list-ul a fost introdusă de William Pugh în anul 1989 ca o alternativă la arbori de căutare echilibrați, precum arborii AVL sau arborii roșu-negru.

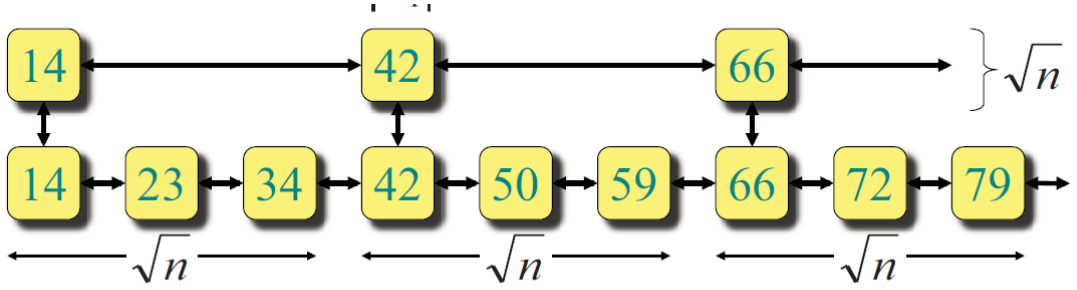
Principalele caracteristici ale skip list-ului sunt:

1. Niveluri multiple: Skip list-ul constă din mai multe niveluri de liste înlănțuite, cu nivelul superior fiind o suprasarcină a nivelului inferior. Fiecare element din skip list este reprezentat de un nod, iar fiecare nod poate avea mai multe referințe către noduri din niveluri superioare.
2. Probabilitatea de nivelare: Nodurile din skip list sunt nivelate în funcție de o probabilitate dată. De exemplu, un nod poate avea o referință către nodul următor din nivelul superior cu o probabilitate de 0,5. Aceasta înseamnă că skip list-ul are o structură probabilistică, iar eficiența operațiilor depinde de probabilitățile de nivelare utilizate.
3. Operații de bază: Skip list-ul oferă operații de bază, precum inserare, ștergere și căutare. Aceste operații se desfășoară în timp O(log n), unde n reprezintă numărul de elemente din skip list. Deoarece skip list-ul nu necesită echilibrarea ca arborii de căutare echilibrați, aceste operații sunt mai ușor de implementat și de înțeles.
4. Eficiență în căutare: Skip list-ul oferă căutare eficientă în timp O(log n) datorită structurii sale de niveluri multiple. Comparativ cu alte structuri de date ordonate, cum ar fi arborii de căutare echilibrați, skip list-ul poate oferi performanțe similare în ceea ce privește căutarea.

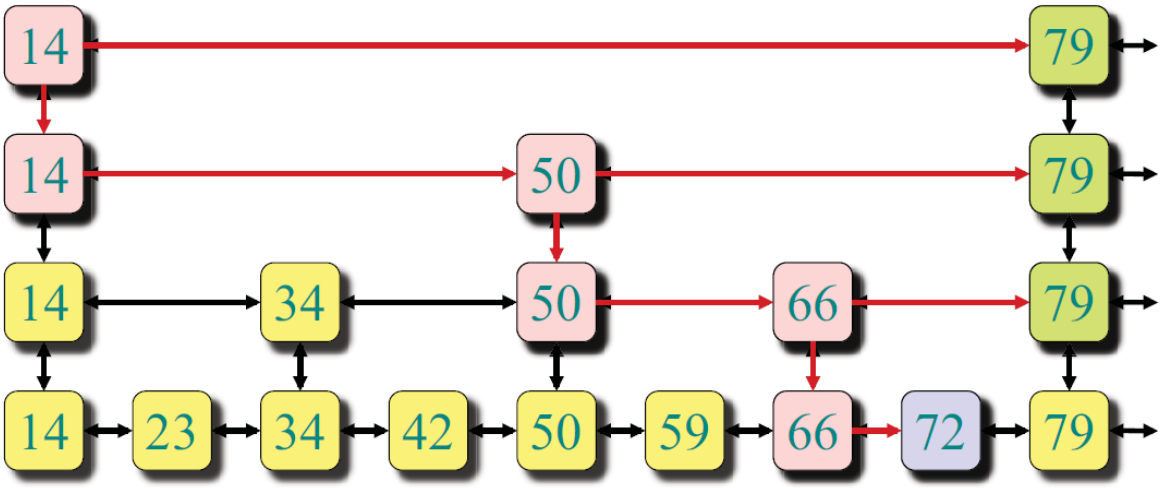
Skip list-ul este utilizat în diverse aplicații care necesită acces rapid la datele ordonate, cum ar fi bazele de date, implementările de liste, seturile ordonate și multe altele. Este o alternativă interesantă la alte structuri de date și oferă un echilibru între complexitatea implementării și eficiența operațiilor de bază. Cu toate acestea, deoarece skip list-ul implică o componentă probabilistică, performanța sa poate varia în funcție de factorii de nivelare aleși.

Pentru o listă normală, sortată, complexitatea căutării unui element este O(n).

Skip Lists presupun niște ”checkpoint”-uri care să faciliteze căutarea, adică vom avea căutare în timp O(√ n).



Skip Lists se pot expanda asemenea unui arbore, astfel încât pe primul nivel se află extremitățile mulțimii, iar pe ultimul întreaga mulțime.
Exemplu: Căutarea numărului 72



5.2. Bloom Filters

”Bloom Filter” este o structură de date probabilistică eficientă din punct de vedere spațial, care este utilizată pentru a testa dacă un element este membru al unei mulțimi.
De exemplu, verificarea disponibilității numelui de utilizator este chiar problema apartenenței, unde setul este lista tuturor numelui de utilizator înregistrat.

Spre deosebire de un tabel hash standard, un filtru Bloom de dimensiuni fixe poate reprezenta un set cu un număr arbitrar de mare de elemente.

Adăugarea unui element nu eșuează niciodată. Cu toate acestea, rata fals pozitivă crește constant pe măsură ce elementele sunt adăugate până când toți biții din filtru sunt setați la 1, moment în care toate interogările dau un rezultat pozitiv.

Filtrele Bloom nu generează niciodată rezultate fals negative, adică să spună că un nume de utilizator nu există atunci când există de fapt.

Prețul pe care îl plătim pentru eficiență este că este de natură probabilistică, ceea ce înseamnă că ar putea exista unele rezultate fals pozitive. Fals pozitiv înseamnă, s-ar putea spune că dat numele de utilizator este deja luate, dar de fapt nu este.
”Bloom Filter” este o structură de date probabilistică eficientă din punct de vedere spațial, care este utilizată pentru a testa dacă un element este membru al unei mulțimi.

Practic, bloom filters înseamnă folosirea mai multor funcții de hash pentru a stoca dacă un element apare într-o mulțime.

Dacă la un vector normal de frecvență, atunci când un oarecare i apare in mulțime, setăm v[i] = 1, aici vom avea k funcții de hash, iar atunci când i apare, vom seta
v[f1(i)] = 1, v[f2(i)] = 1, … v[fk(i)] = 1.
De exemplu, verificarea disponibilității numelui de utilizator este chiar problema apartenenței, unde setul este lista tuturor numelui de utilizator înregistrat.

Spre deosebire de un tabel hash standard, un filtru Bloom de dimensiuni fixe poate reprezenta un set cu un număr arbitrar de mare de elemente.
Filtrele Bloom nu generează niciodată rezultate fals negative, adică să spună că un nume de utilizator nu există atunci când există de fapt.
Prețul pe care îl plătim pentru eficiență este că este de natură probabilistică, ceea ce înseamnă că ar putea exista unele rezultate fals pozitive. Fals pozitiv înseamnă, s-ar putea spune că dat numele de utilizator este deja luate, dar de fapt nu este.

Cum alegem matematic numărul de funcții de hash?

- m – array size
- k – numărul de funcții de hashing
- n - numărul de elemente de inserat

Probabilitatea de false positive:

$$P = (1 - \left(1 - \frac{1}{m}\right)^{kn})^k$$

Pentru o probabilitate de false pozitive *p* și un număr de *n* elemente inserate, atunci lungimea șirului trebuie un șir de lungime

$$m = -\frac{n\ lnP}{(\ln 2)^2}$$

Dat fiind m și n, numărul optim de funcții de hashing este

$$k = \frac{m}{n} ln2$$

Bibliografie:

1. educative.io/blog/data-structures-algorithms
2. irinaciocan.ro
3. "Introduction to Algorithms" de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
4. btechsmartclass.com/data_structures
5. Algoritmi Avansați 2023 c-13 Randomized Data Structures: Skip Lists; Bloom Filters - Lect.Dr. Ștefan Popescu
6. en.wikipedia.org/wiki/Trie
7. en.wikipedia.org/wiki/Heap_(data_structure)