Analiza implementacije projektnih zadataka

[P-2.33] Prvi izvod polinoma

1. Uvod

U ovom zadatku implementirana je klasa `Polynomial` koja automatizuje proces nalaženja prvog izvoda polinoma unetih u standardnoj algebarskoj notaciji.

2. Reprezentacija podataka i parsiranje

Polinom se interno čuva u listi `_polynomial_list`, gde indeks elementa direktno mapira stepen promenljive `x`.

- **Mehanizam parsiranja**: Korišćen je modul `re` (regex) sa složenim šablonom koji prepoznaje znakove, koeficijente, eksponente i slobodne članove.
- **Normalizacija**: Program uspešno interpretira vrednosti poput `x` kao 1x^1 ili `-x` kao -1x^1.

3. Nalaženje izvoda

Rezultat glavne funkcije `calculate_derivative` je nova instanca klase `Polynomial`, što omogućava konzistentnost podataka i lako nalaženje n-tog izvoda.

4. Diskusija

Najveći izazov ove implementacije je predstavljala transformacija iz tekstualnog oblika u listu i obrnuto. Upotreba regex-a omogućava fleksibilnost unosa, dok reprezentacija u vidu liste omogućava efikasno izračunavanje izvoda u O(n) vremenu, gde je n stepen polinoma.

[P-4.23] Rekurzivna pretraga fajlova

1. Uvod

Implementirana je funkcija `find` koja rekurzivno obilazi stablo sistema datoteka kako bi identifikovala sve instance fajla sa specificiranim imenom, počevši od specificirane putanje.

2. Algoritam i rekurzija

Funkcija koristi **Depth-First Search (DFS)** strategiju za obilazak direktorijuma:

- Za svaki zapis u trenutnom direktorijumu, `os.path.join` formira punu putanju.
- Ako je zapis direktorijum, vrši se rekurzivni poziv koji produbljuje pretragu.
- Ako je zapis fajl, vrši se poređenje imena sa ciljanim parametrom.

3. Diskusija

Rekurzivni pristup prirodno odgovara hijerarhijskoj strukturi fajl sistema. Korišćenjem `os.path` modula, rešenje je nezavisno od operativnog sistema.

[P-6.32] ArrayDeque

1. Uvod

Ovaj zadatak predstavlja kompletnu implementaciju Deque ADT-a koristeći kružni niz.

2. Reprezentacija podataka

Logika se oslanja na tri ključne promenljive:

- `_data`: Lista koja se koristi za smeštanje podataka.
- `_front`: Indeks koji pokazuje na trenutni početak reda.
- `_size`: Brojač elemenata koji razdvaja logičku veličinu od fizičkog kapaciteta niza.

3. Ključne operacije i modularna aritmetika

Vremenska složenost O(1) postignuta je izbegavanjem pomeranja elemenata. Umesto toga, koristi se modularna aritmetika za "kruženje" indeksa:

- `add_first`: `(self._front - 1) % len(self._data)`
- `delete_first`: `(self._front + 1) % len(self._data)`
- `resize`: Kada se dostigne `_size == len(_data)`, niz se udvostručuje, a elementi se re-alignuju počevši od indeksa 0.

[P-8.64] ArrayBinaryTree

1. Uvod

U ovom projektnom zadatku implementiran je apstraktni tip podataka (ADT) binarno stablo korišćenjem reprezentacije zasnovane na nizu. Za razliku od klasične implementacije pomoću povezanih čvorova, ova reprezentacija koristi indeksiranje elemenata u nizu kako bi se implicitno definisali odnosi roditelj–dete.

2. Reprezentacija podataka

Binarno stablo je implementirano pomoću Python liste `_data`, gde:

- koren stabla ima indeks `0`
- levi potomak čvora na poziciji `p` se nalazi na indeksu `2p + 1`
- desni potomak čvora na poziciji `p` se nalazi na indeksu `2p + 2`

Neiskorišćene pozicije u nizu popunjene su vrednošću `None`, što omogućava fleksibilno proširivanje stabla bez potrebe za kontinualnim popunjavanjem svih čvorova. Dodatno, promenljiva `_size` čuva broj validnih elemenata u stablu, ne kapaciteta liste `_data`.

3. Osnovne operacije

Implementirane su sve osnovne operacije koje se očekuju od binarnog stabla:

- pristup korenu (`root`)
- pronalaženje roditelja (`parent`)
- pronalaženje levog i desnog deteta (`left`, `right`)

- provera da li je čvor list ili koren (`is_leaf`, `is_root`)
- računanje dubine čvora (`depth`)
- računanje visine stabla (`height`)

Operacija `depth` je implementirana rekurzivno, oslanjajući se na relaciju između čvora i njegovog roditelja, dok se visina stabla računa pomoću pomoćne metode `subtree_height` koja rekurzivno računa visinu stabla od prosleđenog čvora. Shodno tome, metoda `height` poziva metodu `subtree_height` sa argumentom koji predstavlja poziciju korena.

4. Ažuriranje

Dodavanje elemenata (`add_root`, `add_left`, `add_right`) zahteva proveru validnosti pozicije i postojanja odgovarajućeg deteta, čime se sprečava narušavanje strukture stabla.

Brisanje čvora (`delete`) je ograničeno na čvorove sa najviše jednim detetom. U slučaju da čvor ima jedno dete, podstablo se "podiže" na poziciju obrisanog čvora. Ovo je realizovano rekurzivnom metodom `_move_subtree`, koja kopira strukturu podstabla na novu poziciju u nizu.

5. Operacija attach

Metoda `attach` omogućava "kačenje" dva postojeća binarna stabla kao levog i desnog podstabla lista `p`. Kopiranje podstabala realizovano je metodom `_copy_subtree`, koja koristi red kako bi se očuvala struktura stabla prilikom premeštanja elemenata u novi niz.

6. Diskusija

Array-based reprezentacija binarnog stabla omogućava vrlo jednostavan i brz pristup roditeljskim i dečjim čvorovima u vremenskoj složenosti O(1). Međutim, ova prednost dolazi uz potencijalno veće zauzeće memorije. Implementacija pokazuje da je ova reprezentacija naročito pogodna za potpuna i skoro potpuna binarna stabla, dok u slučaju proizvoljne strukture može doći do značajnog broja `None` elemenata u nizu.

[P-12.56] Nerekurzivni In-Place Quick Sort

1. Uvod

Implementirana je optimizovana verzija Quick Sort algoritma koja koristi `stack` strukturu podataka za simuliranje rekurzije.

2. Algoritam i particionisanje

Algoritam se oslanja na **in-place** zamenu elemenata:

- **Pivot**: Bira se poslednji element u trenutnom opsegu.
- **Particionisanje**: Dva pokazivača (`left`, `right`) se kreću jedan ka drugom, menjajući elemente tako da svi manji od pivota završe levo, a veći desno.

3. Simulacija rekurzije

Umesto rekurzivnih poziva, koristi se Python lista kao **stack** koji čuva **tuple**-ove `(a, b)`, granice nesortiranih podnizova. Algoritam radi dokle god stack nije prazan, što garantuje da će ceo niz biti obrađen.