Adatszerkezetek és algoritmusok 2. házi feladat
Interval Tree

Bevezető

A tárgy keretein belül vett adatszerkezetek "tankönyvi példák", amik sokszor alkalmazhatóak való életbeli feladatok esetén is. Némely probléma hatékony megoldásához azonban egy kis kreativitás szükséges: egy-egy tankönyvi adatszerkezetet ki lehet bővíteni extra információkkal, amelyeket tárolni és karbantartani kell, valamint definiálni lehet új műveleteket. A kibővítés általában négy lépésen keresztül történik:

1. A kiindulási adatszerkezet megválasztása
2. A pluszba tárolni és fenntartani kívánt adatok meghatározása
3. A tárolt adatok az eredeti adatszerkezet módosító műveletei során való fentarthatóságának vizsgálata
4. Új műveletek fejlesztése

Interval Tree

A piros-fekete fák önmagukban is elterjedt és használt adatszerkezetek (például az STL-ben található `std::map` és `std::set` is egy piros fekete fa), de előfordulhat, hogy plusz információk tárolásáva tovább javítható a teljesítmény. Vegyünk példának egy eseményeket tároló adatbázist. Minden adat az adatbázisban egy egyedi intervallumból és egy cimkéből áll. Az intervallumokat két egész szám határozza meg a kezdeti időpont és a végpont, ezek az intervallumok határozzák meg, hogy mettől meddig tartott az adott esemény. Egy piros-fekete fát használva könnyen meg tudunk válaszolni olyan kérdéseket, hogy egy adott intervallum szerepel-e az adatbázisban, esetleg mi történt akkor, de ezeket a kérdéseket csak pontos egyezések esetén tudjuk megválaszolni. Mi van, ha arra vagyunk kíváncsiak, hogy egy adott idősáv alatt történt-e vagy folyamatban volt-e, elkezdődött-e egy esemény. Tehát nem pontos intervallum egyezésekre, hanem metsző intervallumokra akarunk rákérdezni. Első ránézésre ebben az esetben nem, vagy csak nehézkesen tudnánk használni egy PF-fát ennek kezelésére. A következő fejezetben leírunk egy módosított PF-fát, amely esetén pluszba tárolt adatok segítségével hatékonyan tudjuk megválaszolni az előbbi kérdést. Sőt azt is megmutatjuk, hogy a plusz infók karbantartása nem fogja módosítani a PF-fa megszokott $\mathcal{O}(log_2 n)$ teljesítményt.

Intervallumok kezelése

Egy $[t_1, t_2]$ zárt intervallum reprezentálására vegyünk egy i objektumot, aminek két adattagja van az $i.low = t_1$ és az $i.high = t_2$. Ekkor azt mondjuk, hogy két intervallum átfedésben van, ha $i \cap i' \neq \varnothing$ azaz $i.low \leq i'.high$ és $i'.low \leq i.high$. Könnyen látható, hogy két intervallum így három állapotban lehet:

- i és i' átfednek
- i balra található i'-től azaz $i.high < i'.low$
- i jobbra található i'-től azaz $i.low > i'.high$

Az `interval tree` egy olyan piros fekete fa, ami minden csúcsában egy-egy ilyen intervallumot tárol és a következő műveleteket tudja:
`INTERVAL-INSERT(T; x)`: A `T` fába beszúrja a `x` intervallumot.
`INTERVAL-DELETE(T; x)`: A `T` fából törli a `x` intervallumot.
`INTERVAL-SEARCH(T; i)`: Visszaad egy mutatót egy `x` elemre a fában, amelyre teljesül, hogy `x` és `i` átfednek, ha nincs ilyen elem, akkor a visszaadott mutató az üres levélelemre mutat.

Gyors viszatekintés:

1. A kiindulási adatszerkezet: Piros-Fekete fa
2. A pluszba tárolni és fenntartani kívánt adatok: az intervallumokon kívül segéd infónak felvesszük a `max` értéket minden csúcshoz, ami az adott részfában található intervallumok végpointjainak maximuma lesz.

3. A tárolt adatok az eredeti adatszerkezet módosító műveletei során való fentarthatóságának vizsgálata: A `max` értékének frissítése egy x csúcsra (a csúcsban tárolt intervallumot $x.int$ jelöli) $\mathcal{O}(1)$ alatt végrehajtható, hiszen:

$$x.max = max(x.int.high, x.left.max, x.right.max)$$

Ebből következően ezt az értéket beszúráskor a lefelé úton minden érintett csúcsnál tudjuk frissíteni, illetve törléskor a törölt csúcstól a gyökérig vezető úton is frissítenünk kell (ez összesen $\mathcal{O}(log_2 n)$ művelet a fa magasságából adódóan). Tehát a beszúrás és törlés nagyságrendileg nem változik. Az utánuk történő kiegyensúlyozásban minden forgatás után két-két csúcsnál kell frissítenünk, így ennek a komplexitása sem változik.

4. Új műveletek: Interval-Search: Egy adott részfában található összes intervallumról el tudjuk dönteni, hogy balra vannak-e az adott intervallumtól (hiszen ha a max kisebb, mint az intervallum bal oldala, akkor minden a részfában található intervallum balra lesz az intervallumtól). Ez alapján az `INTERVAL-SEARCH` egyszerűen olyan lesz, mint a fában keresés, ha van potenciális metszés a bal részfában arra haladunk tovább, ha nincs, azaz a részfa balra van a keresett intervallumtól, akkor jobbra lépünk tovább. Az algoritmus akkor áll le, ha talált egy átfedő intervallumot, vagy eléri a levél elemeket. Mivel minden szinten a döntés $\mathcal{O}(1)$ műveletet igényel ezért a műveletigénye a keresésnek a fa magasságával lesz arányos, azaz $\mathcal{O}(log_2 n)$ lesz.

Kibővített Interval Tree (80p)

A feladat egy `interval tree` implementálása lesz, ami minden intarvallumhoz tárol egy cimkét is. A fában minden intervallum csak egyszer szerepelhet tehát az intervallumot hasnzáljuk úgy, mint órán a fába beszúrt elemeket, ez lesz a kulcs, ez alapján lesznek rendezve az elemek a fában. A tárolt címke nem feltétlenül egyedi, nem vesz részt a rendezésben. A fentiek alapján készítsd el az `interval_tree` osztályt a megfelelő műveletekkel. **Figyelem:** az órai kódhoz képest a `find` is változik, hiszen nem egy bool-lal hanem egy pointerrel fog visszatérni. Az órai kódhoz hasonlóan a `validate` függvényt is írjátok meg az új fa típusra, ez ugyanúgy a binkerfa és a piros fekete tulajdonságokat ellenőrizze. Figyelj, a `max` érték karbantartására az adatszerkezetet módosító műveletek és a kiegyensúlyozás során. Az intervallumokat a fán belül az egyszerűség kedvéért `std::pair<int, int>`-ekként fogjuk reprezentálni. Azaz az eddigi jelölésben az $i.low$-t $i.first$-ként, míg az $i.high$-t $i.second$-ként érhetjük majd el. A `max` értékek ellenőrzésére legyen egy max fügvény ami egy kulcsot (intervallumot) kap paraméterül és a intervallumot tartalmazó csúcshoz tartozó részfának a maximum értékét adja vissza, ha nincs a fában a kulcs, akkor 0-t.

Műveletek - API

A feladat a következő osztály implementálása. Az `interval_search`-n kívül minden művelet úgy működik, mint egy hagyományos PF-fában a max érték karbantartásával kiegészítve. A `find` és az `interval_search` `nullptr`-rel tér vissza, ha az algoritmus eljut a levélig.

Megjegyzés: A find nem `bool`-al tér vissza, hanem egy pointerrel egy csúcs által tárolt kulcs-érték párra.

```
class interval_tree {
public:
  // Konstruktor és destruktor
  interval_tree();
  ~interval_tree();

  // Másoló konstruktor és operátor implementálása opcionális
  interval_tree(const interval_tree &t) = delete;
  interval_tree &operator=(const interval_tree &t) = delete;
  interval_tree(const interval_tree &&t) = delete;
  interval_tree &operator=(interval_tree &&t) = delete;

  // Alapműveletek
  size_t size() const;
  void insert(const std::pair<int, int> &k, const std::string &v);
  void remove(const std::pair<int, int> &k);
```

```
  std::pair<const std::pair<int, int>, std::string> *
    find(const std::pair<int, int> &k);
  const std::pair<const std::pair<int, int>, std::string> *
    find(const std::pair<int, int> &k) const;
  // Új művelet
  std::pair<const std::pair<int, int>, std::string> *
    interval_search(const std::pair<int, int> &i);
  // Ellenőrző fgvk
  void validate() const;
  int max(const std::pair<int, int> &i) const;
};
```

Megjegyzés: a `std::pair<const std::pair<int, int>, std::string>`-nek nincs másoló konstruktora, ezért a törléskor előfordulhat, hogy az értéket nem tudod kicserélni. Ilyenkor a `node`-ok kicserélésével megkerülheted ezt a problémát.

Megjegyzés2: Természetesen minden itt mutatott kód kiegészíthető más függvényekkel, adattagokkal, stb.

Megjegyzés3: a `std::pair`-nek van kisebb operátora, és pont azt csinálja, ami alapján rendezni kell az intervallumokat.

Iterátor API (20p)

Mivel az adatbázist c++-hoz szokott emberek fogják használni, ezért szeretnénk azt STL-hez hasonló API-t biztosítani. Ezért az adatszerkezetünket ellátjuk a következő iterátor osztállyal és kiegészítjük a megfelelő függvényekkel.

```
class _interval_tree {
public:
  // ...
  class iterator {
  public:
    explicit iterator(node *);
    // iteratorok összehasonlítása
    bool operator==(const iterator &);
    bool operator!=(const iterator &);
    // Klasszikus iterátoraritmetika. Vegyük észre, hogy nagyon hasonló a
    // pointeréhez
    iterator &operator++();   // Prefix következőre léptetés iteratorokra
    iterator operator++(int); // Postfix következőre léptetés iteratorokra
    iterator &operator--();   // Prefix előzőre léptetés iteratorokra
    iterator operator--(int); // Postfix előzőre léptetés iteratorokra
    // Dereferencia iterátorokra
    std::pair<const std::pair<int, int>, std::string> &operator*();
    // Structure dereference operátor
    std::pair<const std::pair<int, int>, std::string> *operator->();
  };
  // iteratorral kapcsolatos muveletek
  iterator begin();
  iterator end();
  iterator find_i(const std::pair<int, int> &k);
  // Új művelet
  iterator interval_search_i(const std::pair<int, int> &i);
};
```

Megjegyzés: fa iterátorra láthattok példát a binkerfás advanced kódban a wikin. De az egy konstans iterátor, míg ezen keresztül lehet módosítani az intervallumokhoz tartozó cimkéket. Illetve a PF-fa implementációnkban nincsenek `nullptr`-ek, hanem mindenhova az `empty_leaf` van bekötve.

Példa

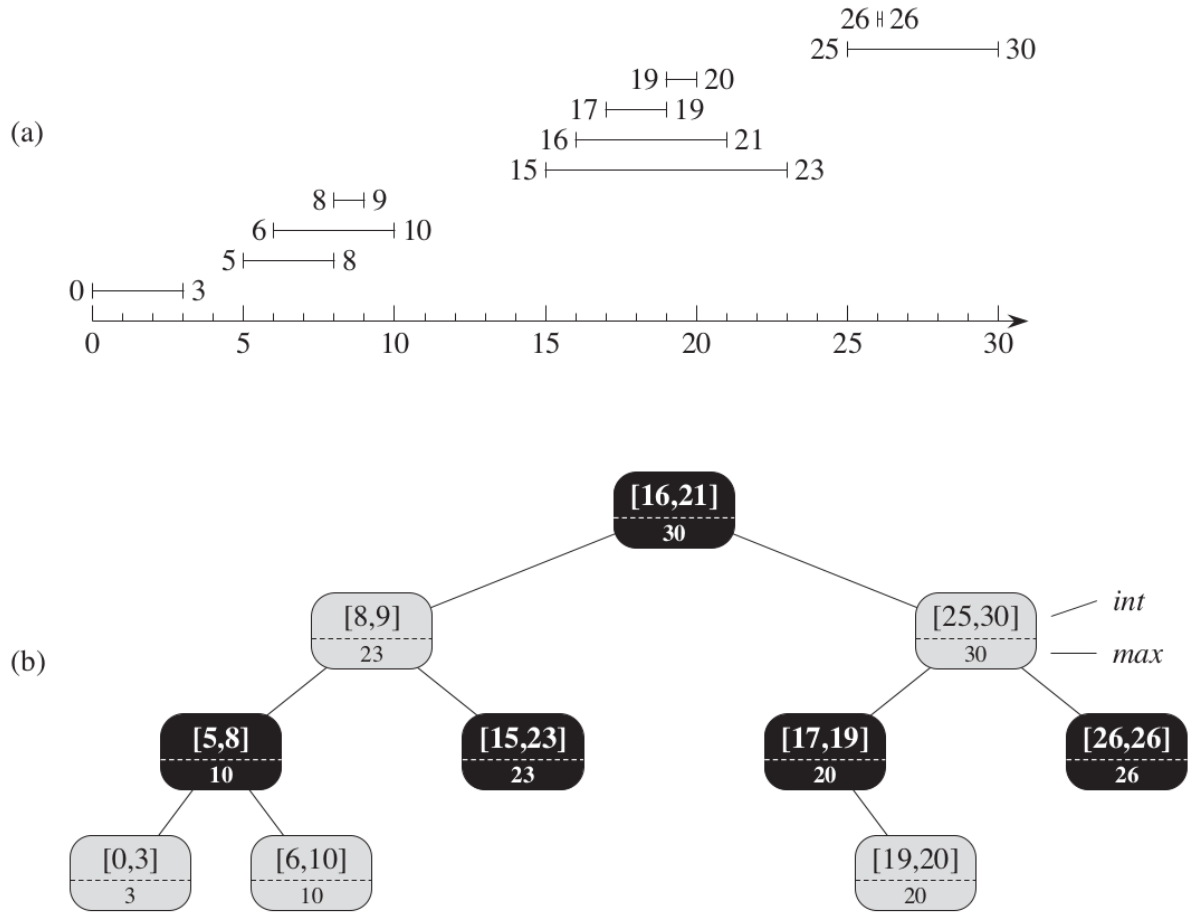

Figure 1: Egy `interval_tree`. Az **a)** ábrán a beszúrt intervallumok, a **b)** ábrán a belőlük felépített fa látható. Egy adott node tartalmazza az intervallumot és a szaggatott vonal alatt pedig a részfában található intervallumok maximális végpontja [1].

A 1. ábrán látható fa esetén például a $i = [14, 15]$ intervallum esetén az interval search a $[16, 21] \rightarrow [8, 9] \rightarrow [15, 23]$ útvonalon haladva a $[15, 23]$ csúcshoz intervallum-cimke párra mutató pointerrel (vagy iterátorral) fog visszatérni. Ezzel szemben a $i = [12, 14]$ intervallumra ugyanazon az útvonalon haladva végül egy `nullptr`-rel vagy `end` iterátorral fog visszatérni.

Hasznos olvasmány, forrás

Az intervallum fák és a PF-fa kibővítéséről részletesebb, talán érthetőbb leírást, példákat és mindent mi szem-szájnak ingere találhattok a Cormen könyvben[1].

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2009, Ch. 14.2-3, pp. 345–355 (2009).
URL http://thuvien.thanglong.edu.vn:8081/dspace/bitstream/DHTL_123456789/3760/2/introduction-to-algorithms-3rd-edition.pdf

Egyéb hasznos infók

API egyben, rövidítésekkel

```
class interval_tree {
public:
  // típus nevek rövidítés érdekében
  // innentol a std::pair<int, int> helyett irhatunk Key_t-t
  using Key_t = std::pair<int, int>; // intervallum tipus
  // a pair<const pair<int, int>, string> helyett meg value_type-t
  // ez kb egy pair<intervallum, cimke> ahol a kulcs ay intervallum a cimke meg
  // csak plusz ino
  using value_type = std::pair<const Key_t, std::string>;

  // Konstruktor és destruktor
  interval_tree();
  ~interval_tree();

  // Másoló konstruktor és operátor implementálása opcionális
  interval_tree(const interval_tree &t) = delete;
  interval_tree &operator=(const interval_tree &t) = delete;
  interval_tree(const interval_tree &&t) = delete;
  interval_tree &operator=(interval_tree &&t) = delete;

  // iterator API
  class iterator {
  public:
    explicit iterator(node *);
    // iteratorok összehasonlítása
    bool operator==(const iterator &);
    bool operator!=(const iterator &);
    // Klasszikus iterátoraritmetika. Vegyük észre, hogy nagyon hasonló a
    // pointeréhez
    iterator &operator++();   // Prefix következőre léptetés iteratorokra
    iterator operator++(int); // Postfix következőre léptetés iteratorokra
    iterator &operator--();   // Prefix előzőre léptetés iteratorokra
    iterator operator--(int); // Postfix előzőre léptetés iteratorokra
    value_type &operator*();  // Dereferencia iterátorokra
    value_type *operator->(); // Structure dereference operátor
  };

  iterator begin();
  iterator end();

  // Alapműveletek
  size_t size() const;
  void insert(const Key_t &interval, const std::string &label);
  void remove(const Key_t &k);
  value_type *find(const Key_t &k);
  const value_type *find(const Key_t &k) const;
  iterator find_i(const Key_t &k) const;
  // Új művelet
  value_type *interval_search(const Key_t &i);
  iterator interval_search_i(const Key_t &i);
  // Ellenőrző fguk
  void validate() const;
  int max(const Key_t &i) const;
};
```