Prirodno-matematički fakultet u Nišu

Heavy-light dekompozicija

Studijski istraživački rad

Student:

Ivan Stošić

Profesor:

Marko Milošević

Niš
Jul, 2019.

Sadržaj

1 Uvod

Heavy-light dekompozicija je tehnika koja se primenjuje na stabla kako bi se ubrzale operacije koje se odnose na puteve u tom stablu. Ideja je da se stablo particioniše na puteve tako da budu zadovoljeni određeni kriterijumi.

1.1 Uvodne definicije

Pre nego što definišemo Heavy-light dekompoziciju, definišimo neophodne osnovne pojmove iz teorije grafova.

Definicija 1.1 *Graf je uređeni par (V, E), gde je V skup čvorova grafa, a E je skup grana.*

U praksi, posebno kod implementacija grafovskih algoritama za skup V se uzima $\{0, 1, \ldots, n-1\}$ za neki nenegativan ceo broj n. U zavisnosti od vrste grafa, skup E može sadržati različite vrste elemenata.

Definicija 1.2 *Usmeren graf je graf kod kojeg je skup grana E podskup skupa $\{(x, y)|x, y \in V, x \neq y\}$.*

Ako je (x, y) grana usmerenog grafa kažemo da je grana polazi iz čvora x i završava se u čvoru y.

Definicija 1.3 *Kod neusmerenog grafa, skupa grana E je podskup skupa $\binom{V}{2} = \{\{x, y\}|x, y \in V, x \neq y\}$.*

Kod neusmerenog grafa kažemo da grana $\{x, y\}$ povezuje čvorove x, y. Svakom čvoru grafa možemo pridružiti nenegativan ceo broj – njegov stepen, odnosno broj grana koje sadrže taj čvor. Kod usmerenih grafova se definišu i ulazni i izlazni stepen – broj grana koje se završavaju i broj grana koje počinju nekim čvorom, redom. List je čvor stepena 1.

Definicija 1.4 *Šetnja u grafu $G = (V, E)$ je niz čvorova $v_1, v_2, \ldots, v_k \in V$ takav da za svaka dva uzastopna čvora v_i, v_{i+1} važi da je $(v_i, v_{i+1}) \in E$ kod usmerenog grafa, odnosno $\{v_i, v_{i+1}\} \in E$ kod neusmerenog grafa.*

Dužinu šetnje ćemo definisati kao broj čvorova u njoj umanjen za jedan, odnosno, kao broj grana.

Definicija 1.5 *Put je šetnja kod koje su svi čvorovi različiti.*

Definicija 1.6 *Čvor y je dostižan iz čvora x ukoliko postoji šetnja koja počinje čvorom x a završava se čvorom y.*

Nije teško pokazati da je ova definicija ekvivalentna onoj gde se zahteva da postoji put, a ne šetnja, koji počinje u čvoru x a završava se u čvoru y. Kod neusmerenih grafova, jasno je da ako je čvor y dostižan iz x da je tada i čvor x dostižan iz y. Tada kažemo i da su čvorovi povezani.

Teorema 1.7 *Relacija povezanosti kod neusmerenog grafa je relacija ekvivalencije.* □

Definicija 1.8 *Neusmeren graf je povezan ako je svaki čvor dostižan iz svakog drugog čvora.*

Definicija 1.9 *Prost ciklus u neusmerenom grafu je šetnja dužine bar 3 kod koje su prvi i poslednji čvor jednaki, dok su svi ostali čvorovi međusobno različiti i različiti od prvog čvora.*

Razlog zašto se u ovoj definiciji zadaje uslov da šetnja mora biti dužine bar 3 jeste taj što se šetnja dužine 2 oblika u, v, u za $u \neq v$ ne smatra prostim ciklusom. Ekvivalentna definicija bi bila da se umesto dužine 3 zahteva da sve iskorišćene grane budu međusobno različite.

Prethodna definicija nam omogućava da uvedemo pojam stabla.

Definicija 1.10 *Stablo je povezan neusmeren graf bez prostih ciklusa.*

Postoji još nekoliko ekvivalentnih definicija stabla.

Teorema 1.11 *Svaki povezan neusmeren graf sa n čvorova i $n-1$ granom je stablo.* □

Teorema 1.12 *Svaki neusmeren graf takav da između svaka dva čvora postoji jedinstven put je stablo.* □

Da bismo definisali hijerarhiju čvorova u stablu, potrebno je da izaberemo jedan čvor, koji ćemo zvati korenom stabla. Često se izbor korena prirodno nameće – na primer, ukoliko se radi o sintaksnom stablu neke rečenice, koren će biti čvor koji predstavlja celu tu rečenicu. Ukoliko to nije slučaj, jedan čvor se proizvoljno bira za koren. Sada možemo definisati pojmove roditelja, dece, predaka, potomaka i podstabala u stablu.

Definicija 1.13 *U stablu $T = (V, E)$ sa korenom r roditelj čvora $x \neq r$ je drugi čvor na jedinstvenom putu od x do r. Za $x = r$ roditelj se ne definiše. Deca čvora x su čvorovi kojima je x roditelj.*

Definicija 1.14 *Preci čvora x su svi čvorovi na putu od x do r. Potomci čvora x su svi čvorovi kojima je on predak.*

Definicija 1.15 *Podstablo čvora x je indukovani podgraf nad skupom potomaka x. Sa $s(x)$ označavamo broj čvorova, odnosno veličinu podstabla čvora x.*

Definicija 1.16 *Dubina čvora x, u oznaci $d(x)$, je dužina puta od x do r.*

1.2 Složenost algoritama

Vreme, odnosno broj koraka i količina utrošene memorije tokom izvršenja nekog algoritma zavisi od ulaznih parametara. *Veliko O* notacija nam olakšava opisivanje i izračunavanje ovih funkcionalnih zavisnosti. Neka je u narednim definicijama domen funkcija f, g skup $\mathbb{N}_0$ a kodomen $\mathbb{R}^+ \cup \{0\}$.

Definicija 1.17 *Skup $O(g)$ definišemo kao skup svih funkcija f za koje važi da postoje konstante c i n_0 takve da je $f(n) \leq cg(n)$ za svako $n \geq n_0$.*

Ovu notaciju koristimo kada želimo da opišemo gornju granicu neke funkcije, do na proizvod sa konstantom. Često umesto $f \in O(g)$ pišemo i $f = O(g)$. Problem ove notacije je upravo u tome što samo daje gornju granicu ponašanja neke funkcije. Zato se uvodi Θ-notacija.

Definicija 1.18 *Skup $\Theta(g)$ definišemo kao skup svih funkcija f za koje važi da postoje pozitivne konstante c_1, c_2 i n_0 takve da je $c_1 g(n) \leq f(n) \leq c_2 g(n)$ za svako $n \geq n_0$.*

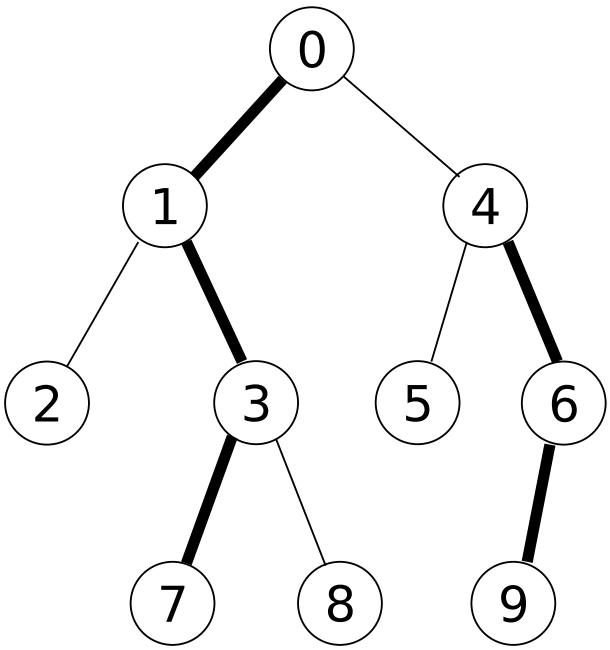
Za algoritam čiji je ulazni parametar n, što može biti broj elemenata niza, broj vrsta matrice, broj čvorova grafa, itd. kažemo da ima vremensku složenost $\Theta(g(n))$ ako je f, gde je $f(n)$ broj elementarnih koraka tokom izvršenja algoritma, u skupu $\Theta(g)$. Slično definišemo memorijsku složenost preko broja iskorišćenih elementarnih memorijskih lokacija npr. bajtova ili bitova.

2 Heavy-light dekompozicija

2.1 Definicija

Neka je $T = (V, E)$ stablo sa korenom $r \in V$. Za svaki čvor x koji ima bar jedno dete definišemo h_x na sledeći način. Neka su $y_1, \ldots, y_k$ sva deca čvora x. Tada je h_x dete čije je podstablo najveće odnosno koje ima najveći broj potomaka. Ukoliko ovaj izbor nije jedinstven, uzima se bilo koje dete.

Posmatrajmo usmeren graf H čiji je skup čvorova V a skup grana je $\{(x, h_x)|x$ ima decu$\}$. Dokažimo da je on jednak uniji puteva. Svaki čvor ima ulazni stepen najviše 1, zato što je jedina grana koja može da ide ka njemu ona koja polazi iz njegovog roditelja. Takođe, jasno je da svaki čvor po definiciji ima izlazni stepen najviše 1. Odavde zaključujemo da se graf može razbiti na komponente, gde je svaka ili put ili ciklus. Ovaj graf ne sadrži cikluse jer se duž svake šetnje u grafu dubina čvora u stablu T povećava. Ovime smo pokazali da je H razbijanje na puteve. Ovo razbijanje upravo zovemo *Heavy-light dekompozicijom* stabla. U stablu T grane (x, h_x) nazivamo teškim, dok sve ostale grane nazivamo lakim granama – odatle potiče i naziv *Heavy-light dekompozicija*.



Primer heavy-light dekompozicije, koren je čvor 0. Teške grane su podebljane.

Ovo razbijanje je od značaja jer zadovoljava sledeću veoma važnu osobinu.

Teorema 2.1 *Neka je T stablo sa n čvorova sa korenom u r a H njegova Heavy-light dekompozicija. Neka je x proizvoljan čvor. Posmatrajmo skup puteva u H kojima pripadaju čvorovi na putu od x do r. Tada ovaj skup sadrži ne više od $\log_2(n) + 1$ elemenata.*

Dokaz. Posmatrajmo put od x do r u T, i neka je taj put $p_1, p_2, \ldots, p_k$, gde je $k = d(x) + 1$. Posmatrajmo dva uzastopna čvora sa tog puta, recimo u, v. Ako je grana $\{u, v\}$ laka, tada je $s(v) \geq 2s(u)$. Zaista, pošto je grana laka, v ima još jedno dete w različito od u čija je veličina podstabla $s(w)$ barem $s(v)$. Pošto je $s(v) \geq s(u) + s(w)$, direktno dobijamo traženu nejednakost. Ako grana nije laka, onda trivijalno važi $s(v) \geq s(u)$. Jasno je da je broj različitih puteva koji sadrže čvorove na putu od x do r u T za jedan veći od broja lakih grana na tom putu, jer svakim prelaskom lake grane dodajemo jedan novi put u H, dok prelaskom teške grane ostajemo na istom putu u H. Dakle, neka je broj lakih grana l. Dokazaćemo da je $l \leq \log_2(n)$. Posmatrajmo niz brojeva $s(p_1), s(p_2), \ldots, s(p_k)$. Pošto je niz neopadajuć, $s(p_1) \geq 1$, a svaka laka grana znači da je sledeći broj bar duplo veći od prethodnog, imamo da je $n = s(r) = s(p_k) \geq 2^l$, odakle direktno dobijamo traženu nejednakost za l. Pošto je broj različitih puteva $l + 1$ ovime kompletiramo dokaz. □.

2.2 Algoritam za konstrukciju

Za skladišvenje grana stabla koristićemo listu susedstva. Konkretno, čvorove označavamo brojevima od 0 do $n - 1$ i pamtimo niz nizova e, gde je $e[x]$ niz svih suseda čvora x. Koristimo šest pomoćnih nizova, svaki dužine n. To su:

- p – roditelj čvora. Služi nam da možemo da razlikujemo roditelja od dece nekog čvora, jer rekurziju radimo samo ka deci.
- h – ukoliko x ima dece, $h[x]$ je njegovo dete ka kojem ide teška grana. U suprotnom $h[x] = -1$.
- d – dubina čvora x je $d[x]$. Za koren je dubina jednaka nuli.
- r – koren, odnosno prvi čvor puta u H kojem pripada čvor x je $r[x]$.

- c – konkatenacija svih puteva u H.
- pos – inverz niza c, odnosno niz koji zadovoljava $pos[c[i]] = i$.

Algoritam se realizuje u dve faze. U prvoj fazi se pretragom u dubinu izračunava veličina svakog podstabla i određuju se teške grane, odnosno vrednosti u nizu h. Pored toga, izračunavaju se vrednosti u nizovima p i d.

Implementacija prve faze algoritma:

```
1  int dfs(int x) {
2    int shx = -1, sx = 1;
3    for (int y : e[x]) {
4      if (y != p[x]) {
5        p[y] = x;
6        d[y] = d[x] + 1;
7        int sy = dfs(y);
8        sx += sy;
9        if (sy > shx) {
10         shx = sy;
11         h[x] = y;
12       }
13     }
14   }
15   return sx;
16 }
```

U drugoj fazi, na osnovu izračunatih vrednosti u nizu h se formiraju putevi, za svaki čvor se pamti koren njegovog puta i takođe se u niz c dodaju čvorovi sa tog puta. Čvor x je koren nekog puta u H ukoliko je on koren celog stabla, ili ako je grana koja spaja njega i njegovog roditelja laka, što važi ukoliko je $h[p[x]] \neq x$.

Implementacija druge faze algoritma:

```
1  void finish() {
2    int n = e.size(), k = 0;
3    for (int i=0; i<n; i++) {
4      if (i == root || h[p[i]] != i) {
5        for (int j=i; j!=-1; j=h[j]) {
6          r[j] = i;
7          c[k] = j;
8          pos[j] = k++;
9        }
10     }
11   }
12 }
```

U drugoj fazi se unutrašnja for-petlja izvršava tačno jednom za svaki čvor, pa je vremenska i memorijska složenost obe faze $O(n)$.

2.3 Primene

2.3.1 k-ti predak čvora

Za čvor x i broj $k \geq 0$, k-ti predak je čvor $p^k(x)$, ukoliko postoji, za šta je dovoljan i potreban uslov da je $k \leq d[x]$. Algoritam za nalaženje ovog pretka je jednostavan. Ukoliko znamo da je taj predak u istom putu u H kao x, što možemo proveriti upoređivanjem dubina čvorova x i $r[x]$, samo se vratimo za k mesta unazad u tom putu. U suprotnom, pretragu nastavljamo od čvora $p[r[x]]$ i k smanjujemo za razliku u dubini.

Implementacija algoritma za nalaženje k-tog pretka:

```
int kth_ancestor(int x, int k) {
  if (d[x] < k)
    return -1;
  while (k > 0) {
    if (k <= d[x] - d[r[x]]) {
      return c[pos[x]-k];
    } else {
      k -= d[x] - d[r[x]] + 1;
      x = p[r[x]];
    }
  }
  return x;
}
```

Složenost algoritma je linearna po broju iteracija, odnosno po broju obiđenih lakih grana. Na osnovu teoreme 2.1 imamo da je vremenska složenost $O(\log n)$.

2.3.2 Najniži zajednički predak

Posmatrajmo dva čvora stabla x i y. Skupovi njihovih predaka se seku u bar jednom elementu – u korenu stabla. Od svih njihovih zajedničkih predaka, posmatrajmo onaj koji ima najveću dubinu. Njega zovemo najnižim ili najdubljim zajedničkim pretkom, u oznaci $LCA(x, y)$ (od *Lowest common ancestor*). Ovaj čvor je jedinstveno određen, jer na putu od bilo kog čvora do korena nijedna dva čvora nemaju istu dubinu. Štaviše, presek puteva od x do korena i od y do korena stabla je upravo put od $LCA(x, y)$ do korena. Za čvor $LCA(x, y)$ važi i da je put od x do y jednak konkatenaciji puteva od x do $LCA(x, y)$ a zatim od $LCA(x, y)$ do y.

U kontekstu Heavy-light dekompozicije, nalaženje najnižeg zajedničkog pretka je korisno i zato što nam omogućava da opišemo put između svaka dva čvora kao uniju malog broja delova puteva iz Heavy-light dekompozicije, o čemu govori sledeća teorema:

Teorema 2.2 *Neka je $T = (V, E)$ stablo, H njegova Heavy-light dekompozicija, i čvorovi $x, y \in V$. Skup puteva u H koji sadrže čvorove na putu u T od x do y sadrži ne više od $2 \log_2(n) + 1$ elemenata.*

Dokaz. Na osnovu teoreme 2.1 znamo da je put od x do korena u T sadržan u ne više od $\log_2(n) + 1$ puteva u H. Isto važi i za put od y do korena. Unija ova dva skupa puteva ima bar jedan element manje, jer je presek tih skupova put u H koji sadrži koren stabla, odnosno ima ne više od $2(\log_2(n) + 1) - 1 = 2\log_2(n) + 1$ elemenata. Kako je put od x do y u T podskup unije puteva od x do r i od y do r, važi i da će taj put da bude sadržan u ne više od $2\log_2(n) + 1$ puteva u H. □

Opišimo sada algoritam za nalaženje LCA za neka dva čvora. Ideja je da približavamo te čvorove korenu stabla, vodeći računa da ne preskočimo stvarnu vrednost LCA. Za to će nam pomoći sledeća tvrđenja:

Teorema 2.3 *Ako je* $r[x] = r[y]$*, tada je* $LCA(x, y) = x$ *ukoliko je* $d[x] < d[y]$*, inače je* $LCA(x, y) = y$*.*

Dokaz. Pošto je $r[x] = r[y]$ ova dva čvora se nalaze na istom putu u H, pa je jedan od njih drugome predak. LCA je čvor na manjoj dubini. □

Teorema 2.4 *Ako je* $r[x] \neq r[y]$ *i* $d[r[x]] \leq d[r[y]]$*, tada je* $LCA(x, y) = LCA(x, p[r[y]])$*.*

Dokaz. Ukoliko je z proizvoljan predak čvora y, tada je $LCA(x, z)$ ili jednako z, ili jednako $LCA(x, y)$, a oba važe samo kad je $z = LCA(x, y)$. Pretpostavimo suprotno, $LCA(x, y) \neq LCA(x, p[r[y]])$. Pošto je $p[r[y]]$ predak čvora y, imamo da je $LCA(x, p[r[y]]) = p[r[y]]$ predak čvora x a samim tim i čvora $LCA(x, y)$. Put od y do $p[r[y]]$ sadrži $LCA(x, y)$, a pošto je $LCA(x, y) \neq p[r[y]]$ onda je $r[y]$ predak čvora $LCA(x, y)$. Posmatrajmo sada put u H koji sadrži x. Taj put ne može da sadrži $LCA(x, y)$ jer on pripada putu čiji je koren $r[y]$, a iz uslova teoreme imamo da je $r[x] \neq r[y]$. Pošto je $LCA(x, y)$ predak čvora x, put u H koji sadrži x mora da se zaustavi pre $LCA(x, y)$, pa je $d[r[x]] > d[LCA(x, y)] \geq d[r[y]]$ što je u kontradikciji sa uslovom teoreme. □

Ovo nam omogućava da u jednoj iteraciji ili zaključimo da je jedan od čvorova LCA, ili da približimo jedan čvor korenu preskačući tačno jednu laku granu.

Implementacija algoritma za nalaženje LCA:

```
1 int lca(int x, int y) {
2   while (r[x] != r[y]) {
3     if (d[r[x]] > d[r[y]])
4       swap(x, y);
5     y = p[r[y]];
6   }
7   return d[x] < d[y] ? x : y;
8 }
```

Složenost algoritma je linearna po broju iteracija, odnosno broju preskočenih lakih grana, što po teoremi 2.2 iznosi ne više od $2\log_2(n) + 1$ tj. $O(\log n)$.

2.3.3 Obrada puteva u stablu

Heavy-light dekompozicija nam omogućava da probleme kod kojih treba na neki način obraditi put u stablu svedemo na sličnu obradu $O(\log n)$ segmenata u nizu. Jedna od osnovnih struktura za obradu segmenata jeste *segmentno stablo*. Ukratko, za niz elemenata $a_0, \ldots, a_{n-1}$ iz monoida $(A, +)$ segmentno stablo omogućava da se u logaritamskoj složenosti izračuna zbir elemenata bilo kog podsegmenta $a_l + a_{l+1} + \ldots + a_r$, kao i da se vrše izmene, odnosno dodele vrednosti elementima niza a. Tako se ove iste operacije mogu podržati na putevima u stablu u vremenskoj složenosti $O(\log^2 n)$.

Kod monoida koji nisu komutativni treba voditi računa o redosledu elemenata, odnosno, za put od u do v, deo puta od u do $LCA(u, v)$ ide ka korenu stabla, odnosno, suprotno od smera puteva u H, dok od $LCA(u, v)$ do v ide u smeru puteva u H, pa je potrebno napraviti još jedno segmentno stablo koje čuva obrnute sume.

Segmentno stablo se inicijalizuje zadavanjem veličine n. Ono može zauzeti do 4 puta više memorije od one potrebne da se skladišti niz sa n elemenata.

Inicijalizacija segmentnog stabla:

```
template<class T>
class segment_tree {
  vector<T> a;
  int maxn;

  segment_tree(int n) {
    maxn = 1;
    while (maxn < n)
      maxn <<= 1;
    a.resize(2*maxn);
  }

  // ...
};
```

Izmena elemenata ima jednostavu iterativnu implementaciju.

Izmena elementa u segmentnom stablu:

```
void update(int x, T y) {
  x += maxn;
  a[x] = y;
  while (x > 1) {
    x >>= 1;
    a[x] = a[2*x] + a[2*x+1];
  }
}
```

Upit za podsegment takođe ima jednostavnu implementaciju, ali je dokaz vremenske složenosti komplikovaniji. Obe operacije imaju vremensku složenost $O(\log n)$.

Upit za sumu podsegmenta u segmentnom stablu:

```
T query(int l, int r, int x, int xl, int xr) {
  if (r < xl || xr < l)
    return T();
  if (l <= xl && xr <= r)
    return a[x];
  int xm = (xl+xr) >> 1;
  return
    query(l, r, 2*x, xl, xm) +
    query(l, r, 2*x+1, xm+1, xr);
}

T query(int l, int r) {
  return query(l, r, 1, 0, maxn-1);
}
```

Konačno, modifikacijom algoritma za nalaženje LCA podržavamo upite na putevima u stablu kao i izmene vrednosti u čvorovima. Sledeća implementacija podrazumeva da je monoid vrednosti komutativan.

Operacije na putevima sa izmenama:

```
segment_tree<T> tree;
//...

void update(int x, T y) {
  tree.update(pos[x], y);
}

T query(int x, int y) {
  T sum = T();
  while (r[x] != r[y]) {
    if (d[r[x]] > d[r[y]])
      swap(x, y);
    sum = sum + tree.query(pos[r[y]], pos[y]);
    y = p[r[y]];
  }
  if (d[x] > d[y])
    swap(x, y);
  return sum + tree.query(pos[x], pos[y]);
}
```

Ukoliko se operacije na putevima odnose na obiđene grane a ne čvorove, možemo za svaku granu stabla $\{u, v\}$ dodati jedan novi čvor (nazovimo ga

w) a zatim zameniti tu granu dvema granama $\{u, w\}, \{w, v\}$. U originalnim čvorovima stabla čuvamo neutrale za monoid vrednosti. Cela implementacija zajedno sa rešenjem zadatka QTREE može se naći u dodatku A.

A Implementacija u jeziku C++

Implementacija je testirana na sajtu SPOJ, na problemu QTREE. Sledi kratak opis problema na kojem je testirana. Potrebno je učitati stablo, gde svaka grana ima pridruženu početnu težinu. Zatim je neophodno učitati određen broj komandi jednog od dva tipa. Kod prvog tipa potrebno je izmeniti težinu grane na neku zadatku vrednost. Kod drugog tipa potrebno je odštampati granu maksimalne težine na putu između neka dva zadata čvora.

Za kompajliranje priloženog koda je dovoljan kompajler koji podržava verziju `c++11`.

Implementacija problema QTREE

```
#include <vector>
#include <iostream>
#include <string>
using namespace std;

struct hld_basic {
  vector<vector<int>> e;
  int root;

  vector<int> p, h, d, r, pos, c;

  void init(const vector<vector<int>>& e_, int root_ = 0) {
    e = e_;
    root = root_;
    int n = e.size();
    p = h = d = r = pos = c = vector<int>(n, -1);

    d[root] = 0;
    dfs(root);
    finish();
  }

  int dfs(int x) {
    int shx = -1, sx = 1;
    for (int y : e[x]) {
      if (y != p[x]) {
```

```
27          p[y] = x;
28          d[y] = d[x] + 1;
29          int sy = dfs(y);
30          sx += sy;
31          if (sy > shx) {
32            shx = sy;
33            h[x] = y;
34          }
35        }
36      }
37      return sx;
38    }
39
40    void finish() {
41      int n = e.size(), k = 0;
42      for (int i=0; i<n; i++) {
43        if (i == root || h[p[i]] != i) {
44          for (int j=i; j!=-1; j=h[j]) {
45            r[j] = i;
46            c[k] = j;
47            pos[j] = k++;
48          }
49        }
50      }
51    }
52
53    int kth_ancestor(int x, int k) {
54      if (d[x] < k)
55        return -1;
56      while (k > 0) {
57        if (k <= d[x] - d[r[x]]) {
58          return c[pos[x]-k];
59        } else {
60          k -= d[x] - d[r[x]] + 1;
61          x = p[r[x]];
62        }
63      }
64      return x;
65    }
66
67    int lca(int x, int y) {
68      while (r[x] != r[y]) {
69        if (d[r[x]] > d[r[y]])
70          swap(x, y);
71        y = p[r[y]];
72      }
73      return d[x] < d[y] ? x : y;
74    }
75  };
```

```
76
77  template<class T>
78  struct segment_tree {
79    vector<T> a;
80    int maxn;
81
82    segment_tree(int n) {
83      maxn = 1;
84      while (maxn < n)
85        maxn <<= 1;
86      a.resize(2*maxn);
87    }
88
89    void update(int x, T y) {
90      x += maxn;
91      a[x] = y;
92      while (x > 1) {
93        x >>= 1;
94        a[x] = a[2*x] + a[2*x+1];
95      }
96    }
97
98    T query(int l, int r, int x, int xl, int xr) {
99      if (r < xl || xr < l)
100       return T();
101     if (l <= xl && xr <= r)
102       return a[x];
103     int xm = (xl+xr) >> 1;
104     return
105       query(l, r, 2*x, xl, xm) +
106       query(l, r, 2*x+1, xm+1, xr);
107   }
108
109   T query(int l, int r) {
110     return query(l, r, 1, 0, maxn-1);
111   }
112 };
113
114 template<class T>
115 struct hld : hld_basic {
116   segment_tree<T> tree;
117
118   hld(const vector<vector<int>>& e, int root = 0)
119     : tree(e.size())
120   {
121     init(e, root);
122   }
123
124   void update(int x, T y) {
```

```
125     tree.update(pos[x], y);
126   }
127
128   T query(int x, int y) {
129     T sum = T();
130     while (r[x] != r[y]) {
131       if (d[r[x]] > d[r[y]])
132         swap(x, y);
133       sum = sum + tree.query(pos[r[y]], pos[y]);
134       y = p[r[y]];
135     }
136     if (d[x] > d[y])
137       swap(x, y);
138     return sum + tree.query(pos[x], pos[y]);
139   }
140 };
141
142 struct max_int {
143   int x;
144
145   max_int(int x = -2e9) : x(x) {}
146
147   explicit operator int() const { return x; }
148
149   max_int operator+ (max_int b) const {
150     return max(x, b.x);
151   }
152 };
153
154 int main() {
155   ios_base::sync_with_stdio(false);
156   cin.tie(nullptr);
157   int t;
158   cin >> t;
159   while (t--) {
160     int n;
161     cin >> n;
162     vector<vector<int>> e(2*n-1);
163     vector<int> costs(n-1);
164     for (int i=n; i<2*n-1; i++) {
165       int u, v;
166       cin >> u >> v >> costs[i-n];
167       u--, v--;
168       e[i].push_back(u);
169       e[u].push_back(i);
170       e[i].push_back(v);
171       e[v].push_back(i);
172     }
173     hld<max_int> a(e);
```

```
174     for (int i=n; i<2*n-1; i++)
175       a.update(i, costs[i-n]);
176     while (1) {
177       string cmd;
178       cin >> cmd;
179       if (cmd == "DONE")
180         break;
181       else if (cmd == "QUERY") {
182         int u, v;
183         cin >> u >> v;
184         u--, v--;
185         cout << (int)a.query(u, v) << '\n';
186       } else {
187         int x, y;
188         cin >> x >> y;
189         a.update(x+n-1, y);
190       }
191     }
192   }
193 }
```

Reference

[1] Laaksonen A. *Guide to Competitive Programming*, Springer International Publishing (2017)

[2] D. D. Sleator , R. E. Tarjan. *A data structure for dynamic trees*. Proceedings of the thirteenth annual ACM symposium on Theory of computing, p.114-122, May 11-13, 1981, Milwaukee, Wisconsin, USA

[3] D. Harel , R. E. Tarjan. *Fast Algorithms for Finding Nearest Common Ancestors*. SIAM Journal on Computing, 1984, Vol. 13, No. 2 : p.338-355

[4] L. Wang, X. Wang. *A Simple and Space Efficient Segment Tree Implementation*. (2018). arXiv:1807.05356

[5] Bender M.A., Farach-Colton M. *The LCA Problem Revisited*. Gonnet G.H., Viola A. (eds) LATIN 2000: Theoretical Informatics. LATIN 2000. Lecture Notes in Computer Science, vol 1776. Springer, Berlin, Heidelberg