ডাটা স্ট্রাকচার কম্পিউটার সাইন্সের চমতকার অংশগুলোর একটি। আমরা অসংখ্য উপায়ে কম্পিউটারে ডাটা জমা রাখতে পারি। আমরা বাইনারি ট্রি বানাতে পারি,পরে সে গাছ বেয়ে বেয়ে logN এ ডাটা বের করে আনতে পারি,বাসের লাইনের মত কিউ বানাতে পারি,প্রিফিক্স ট্রি বা trie বানিয়ে খুব দ্রুত স্ট্রিং সার্চ করতে পারি। আজ আমরা দেখবো অসাধারণ একটি ডাটা স্ট্রাকচার যার নাম “ডিসজয়েন্ট সেট”।

kruskal’s MST বা tarjan’s offline LCA ইত্যাদি অ্যালগোরিদম ইমপ্লিমেন্ট করতে ডিসজয়েন্ট সেট খুব গুরুত্বপূর্ণ। এটি ইমপ্লিমেন্ট করতে আমাদের একটি অ্যারে ছাড়া কিছু লাগবেনা।
প্রথমে আমরা দেখবো কি ধরণের কাজে আমাদের এই ডাটা স্ট্রাকচারটি দরকার। মনে করি A,B,C,D,E ৫ জন মানুষ। A-B যদি বন্ধু হয় এবং B-C যদি বন্ধু হয় তাহলে আমরা বলতে পারি A-C ও বন্ধু,অর্থাত তাদের বন্ধুত্ব transitive। এখন আমি বলে দিলাম যে A-B,B-C,D-E এরা পরস্পরের বন্ধু। এখন তোমাকে প্রশ্ন করলাম A-C কি পরস্পরের বন্ধু? B-D কি পরস্পরের বন্ধু? প্রথম প্রশ্নের উত্তর হবে “হ্যা”,২য় প্রশ্নের উত্তর হবে “না”। আমাদের তথ্যগুলোকে গ্রাফের মাধ্যমে দেখাতে পারি:

সহজেই বোঝা যাচ্ছে গ্রাফে যেসব নোড একই কম্পোনেন্ট বা সাবগ্রাফের মধ্যে আছে তারা পরস্পরের বন্ধু। তাহলে দু-জন ব্যক্তি বন্ধু নাকি সেটা জানতে হলে আমাদের দেখতে হবে তারা একই সাবগ্রাফে আছে নাকি। সহজেই বিএফএস বা ডিএফএস চালিয়ে এটা বের করা যায়। কিন্তু এগুলোর থেকেও ভালো উপায় হলো “ডিসজয়েন্ট সেট”,কেনো এটা ভালো সেটা কিছুক্ষণ পরেই বুঝতে পারবে।

প্রতিটি মানুষ একটি করে নোড হলে ডিসজয়েন্ট সেট দিয়ে সমাধানের আইডিয়া এরকম:

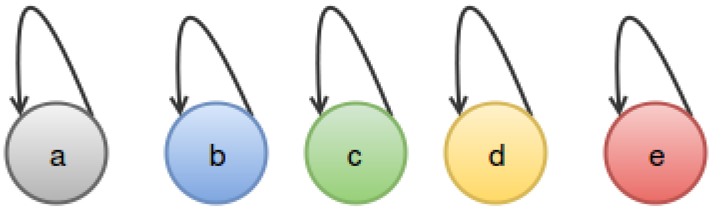
১. *যারা পরস্পরের বন্ধু তারা সবাই একই সেটের অন্তর্গত*

২. *যতগুলো সাবগ্রাফ থাকবে ততগুলো সেট থাকবে*

৩. *প্রতিটি সেটের একটি* representative *থাকবে।*

৪. *দুটি নোডের* representative *একই হলে তারা একই সেটে আছে। সুতরাং দুজন ব্যক্তি বন্ধু নাকি বুঝতে আমাদের তারা যে সেটে আছে তার* representative *চেক করতে হবে।*

ঘোলাটে লাগছে? সমস্যা নেই,আমরা গ্রাফিকালভাবে এখন ব্যাপারটি বুঝবো,সাথে কোডও দেখবো। শুরুতে কেও কারো বন্ধু নয়। তাহলে সবাই আলাদা আলাদা সেটে আছে। এবং যেহেতু সেটগুলোতে মাত্র একটি করে সদস্য তাই সেই সদস্যটিই হবে সেটের representative। ছবিটি দেখো:

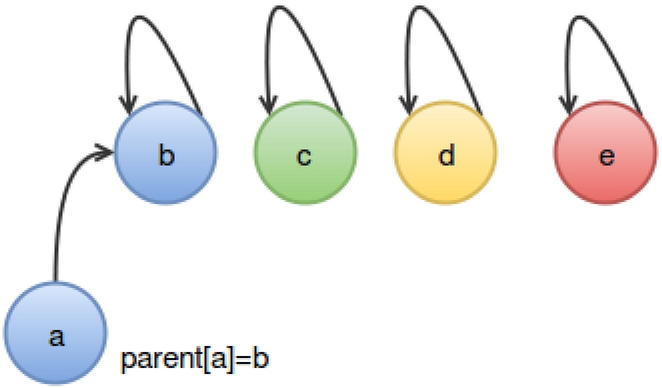


আমরা সেটের representative কে সবসময় **হলুদ রং** দিবো।
এই অংশটি ইম্প্লিমেন্ট করবো এভাবে:

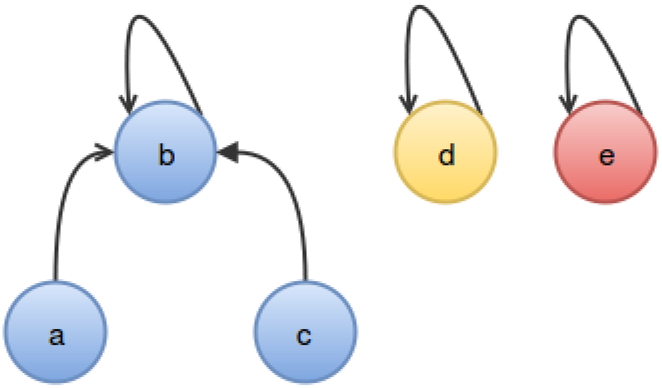
```
#Python2.7.6
#assume a=1,b=2,c=3,d=4,e=5
element=5
par=[None]*(element+1) #Creating an array with 6 elements


def makeset(n):
        par[n]=n

for i in range(1,element+1):
        makeset(i)
```

par নামক অ্যারেটা দিয়েই আমরা সেট বানাবো। শুরুতে সবার প্যারেন্ট সে নিজেই। প্যারেন্ট বলতে বুঝাচ্ছে নোডটি কার সাথে সংযুক্ত। একটি মাত্র সাধারণ অ্যারে ব্যবহার করে আমরা পুরো স্ট্রাকচারটি বানাবো! এটা একধরনের ট্রি স্ট্রাকচার।

এখন a আর b হঠাত ঠিক করলো যে তারা বন্ধু হবে। তাহলে তাদেরকে একই সেটের মধ্য আসতে হবে। a ঠিক করলো যে b এর representative কে নিজের representative বানিয়ে ফেলবে,তাহলেই তারা একই সেটে চলে আসবে। তাই a এসে b এর সাথে যুক্ত হয়ে গেলো কারণ b এর representative হলো b নিজেই। ছবি দেখো:

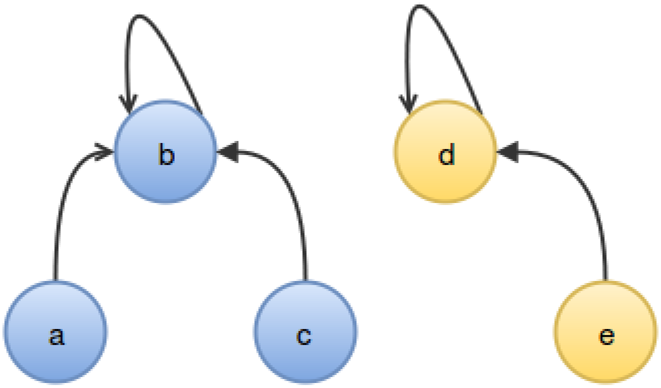


এখন c ভাবলো সেও a এর বন্ধু হবে। এখন c কার নিচে যুক্ত হবে? a এর নিচে c যুক্ত না হবেনা,c যুক্ত হবে a এর representative এর নিচে,তারমানে b এর নিচে।



parent[c]=representative(a)=b

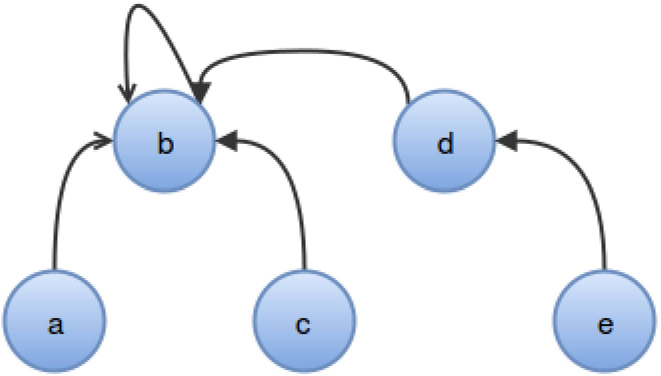
d,e কেও বন্ধু বানিয়ে দেই:



parent[c]=representative(a)=b

পরের ধাপটি ভালো করে দেখো। এবার আমরা e আর c কে বন্ধু বানাবো। c যেই সেটে সেটার representative হলো b। তার সাথে আমরা যুক্ত করবো e এর representative কে। e এর

representative হলো d। b কে d এর প্যারেন্ট বানিয়ে দিলাম। তাহলে b এখন e এরও representative হয়ে গেলো।

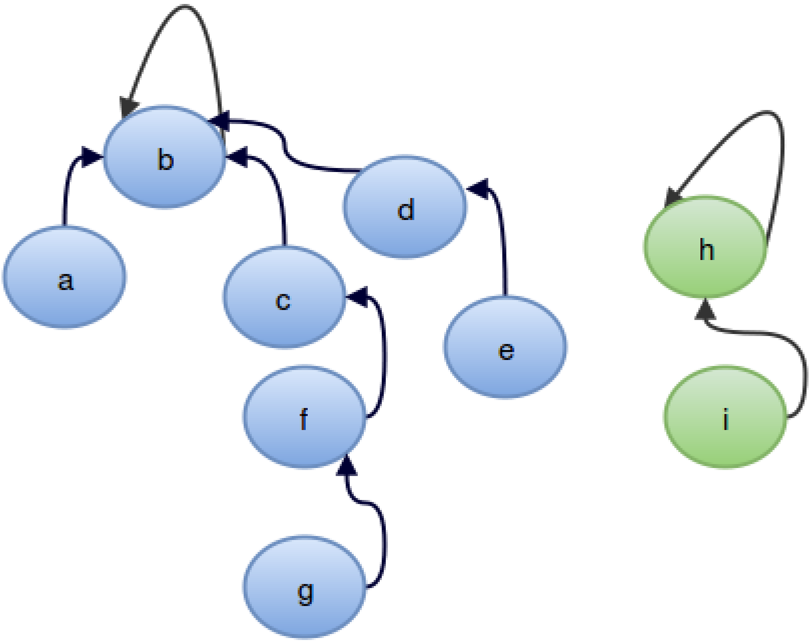


আমরা আবার একটু কোড দেখি:

```
#Python2.7.6
def union(a,b):
    u=find(a)
    v=find(b)
    if u==v:
        print "They are already friends"
    else:
        par[u]=v #Or you can write par[v]=u too
```

আমরা যে দুটি নোডকে বন্ধু বানাবো তাদের representative বের করলাম find() ফাংশন কল করে। (ফাংশনটির ডেফিনেশন পরে দেখবো) এবার একটিকে আরেকটির প্যারেন্ট বানিয়ে দিলাম।

এখন প্রশ্ন হলো representative খুজবো কিভাবে? মনোযোগদিয়ে এই পর্যন্ত পড়ে থাকলে একটা জিনিস চোখে পড়ার কথা,**কোনো একটি সেটের representative element এর প্যারেন্ট সেই এলিমেন্ট নিজেই,অর্থাত এলিমেন্টটি যদি হয় r তাহলে par[r]=r।** আমরা আরেকটু বড় গ্রাফ দেখি। মনে করি a,b,c,d,e এর বন্ধু হতে আরো কয়েকজন চলে এসেছে:

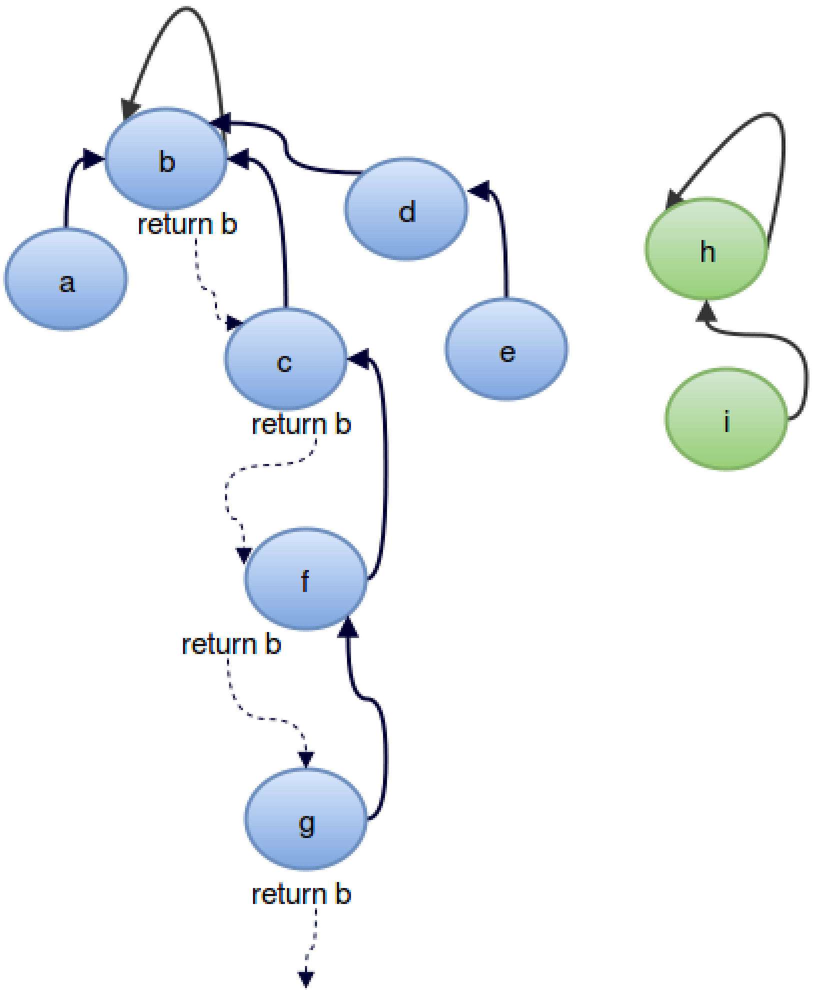


আগের সেটেই শুধু কিছু নোড যুক্ত করা হয়েছে পরের অংশ বোঝার সুবিধার জন্য। ধরি f এর representative বের করতে হবে। f এর প্যারেন্ট সে নিজে নয়,তাহলে তার representative খুজতে তার প্যারেন্টের প্যারেন্ট চেক করি। f এর প্যারেন্ট c। c ও representative নয় কারণ সে নিজেই নিজের প্যারেন্ট নয়,c এর প্যারেন্ট b এবং b নিজেই নিজের প্যারেন্ট। তাহলে b ই হলো representative। ফাংশনটি লিখে ফেলি:

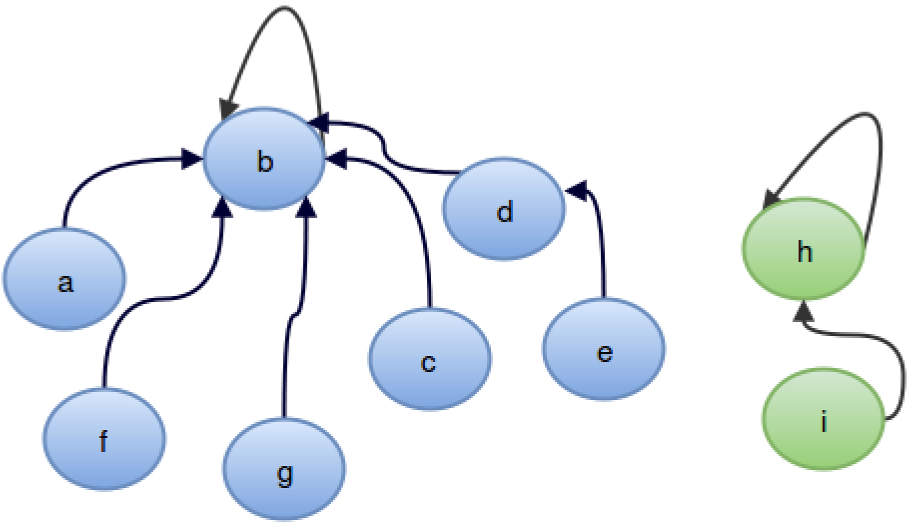
```
#python2.7.6
def find(r):
        if par[r]==r: return r
        return find(par[r])
```

ফাংশনটি গাছ(tree) বেয়ে উপরে উঠলো যতক্ষননা representative কে খুজে পাওয়া যায়। আগের ফাংশনদুটি(union এবং makeset) তাদের আসল কাজ করেছে O(1) এ। find() ফাংশনে এসে complexity'r প্রশ্ন এসে পড়েছে। worst case এ O(n) টাইম লাগতে পারে উপরের find() ফাংশনের,tree এর depth যত বেশি হবে তত টাইম বেশি লাগবে। ভার্সিটির ল্যাবে লেখা কোডের জন্য এটা ঠিক আছে,কিন্তু কনটেস্টে বা অন্য কোথাও যদি নোড অনেক বেশি হয় তাহলে পারফরম্যান্স খারাপ দিবে।

এখানে একটি চমৎকার optimization আছে। আসলে এই optimization টাই ডিসজয়েন্ট সেট স্ট্রাকচারের সৌন্দর্য। প্রতিটি find() ফাংশন কল এর সাথে সাথে আমরা গ্রাফটি এমন ভাবে পরিবর্তন করবো যেন depth কমে যায়। g এর representative খুজতে আমরা g এর প্যারেন্ট f কে কল দিয়েছি। f তার প্যারেন্টকে কল দিয়ে representative খুজে রিটার্ণ করেছে। রিটার্ণ ভ্যালুগুলোকে এভাবে দেখতে পারি:



প্রতিটি নোডের নিচে রিটার্ণ ভ্যালু লিখে দেয়া হয়েছে। প্রতিটি নোড তার আগের ফাংশন একই ভ্যালু রিটার্ণ করে করে,যেটা সবশেষে আমরা পাই,এই ভ্যালুটাই representative। আমরা return find(par[r]) না লিখে যদি আগে লিখতাম par[r]=find(par[r]) এবং তারপর লিখতাম “return par[r]” তাহলে কি ঘটতো? রিটার্ণ করার সময় রিটার্ণ ভ্যালুটাকে নিজের প্যারেন্ট বানিয়ে ফেলতো,অর্থাত সবাই **representative কেই বানিয়ে ফেলতো নিজের প্যারেন্ট!** ফাংশন কলের সাথে সাথে গ্রাফটি হয়ে যেত এমন:



ট্রি এর depth কমে গিয়েছে,তাই নয় কি?এটাকে বলা হয় path compression। পরে আমরা যখন f এর representative খুজবো তখন আর মাত্র একটি ডাল(edge) বেয়ে উঠতে হবে। wiki তে সুন্দর করে লেখা আছে:

path compression, is a way of flattening the structure of the tree whenever Find is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, as Find recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly.

মোটামুটি এই হলো আমাদের disjoint set। দুটি নোড একই সেটে আছে নাকি চেক করতে আমরা তাদের representative বের করে তারা সমান নাকি সেটা চেক করবো। দুটি নোড কে একই সেটে নিতে হলে তাদের representative বের করে একটিকে আরেকটির প্যারেন্ট বানিয়ে দিবো। kruskal এ দুটি নোড একই সাব-ট্রিতে আছে নাকি সেটা চেক করতে disjoint set ব্যবহার করা হয়। একই tree তে না থাকলে union ফাংশনটি কল করে এক করা হয়।
এখন যদি সত্যিই কিছু শিখতে চাও তাহলে ঝটপট কিছু প্রবলেম সলভ করে ফেলো:

Graph connectivity
Nature
Virtual Friend(*)
আরো জানতে চাইলে:

টপকোডার টিউটোরিয়াল
Wikipedia

[সি++ কোডগুলো বদলে পাইথনে লিখে দেয়া হলো, তবে সেজন্য বুঝতে সমস্যা হবার কথা না। সমস্যা হলে যোগাযোগ কর]