۱ مرتب سازی هرمی

با استفاده از الگوریتم Build-Heap می‌توان یک روش دیگر برای مرتب سازی یک آرایه بدست آورد. در این روش ابتدا هرم بیشینه را می‌سازیم. می‌دانیم که بزرگترین عنصر آرایه در ریشه قرار گرفته است. حال عنصر ریشه را حذف می‌کنیم (آن را در مکان $A[n]$ قرار می‌دهیم) و عنصر $A[n]$ را بجای آن در ریشه قرار می‌دهیم. حال انگار یک هرم با $n-1$ عنصر داریم که البته ممکن است ترتیب هرم بیشینه را نداشته باشد (چون ریشه جدید ممکن است از عناصر زیردرختش کوچکتر باشد) اما هرم حاصل را می‌توان با فراخوانی MAX-Heapify برای ریشه درخت اصلاح کرد و ترتیب هرم را به حالت بیشینه در آورد. پس هر بار ریشه حذف و در انتهای آرایه قرار می‌گیرد و طول آرایه یکی کم می‌شود. شبه‌کد زیر روال کار الگوریتم مرتب‌سازی هرمی را نشان می‌دهد.

```
Heap-Sort(A)
1. Build-Heap(A)
2. for i = length(A) downto 2
3.          swap(A[1], A[length(A)])
4.          A.pop()
5.          MAX-Heapify(A,1)
```

مرتب سازی هرمی در بدترین حالت $O(n\log n)$ عمل مقایسه انجام می‌دهد. زمان اجرای Build-Heap از مرتبه $\Theta(n)$ است. از طرفی دیگر الگوریتم بالا زیررویه MAX-Heapify را $n-1$ بار فراخوانی می‌کند. هر فراخوانی هزینه $O(\log n)$ دارد. پس در کل $O(n\log n)$ عمل انجام می‌شود.

۲ مرتب سازی با مقایسه

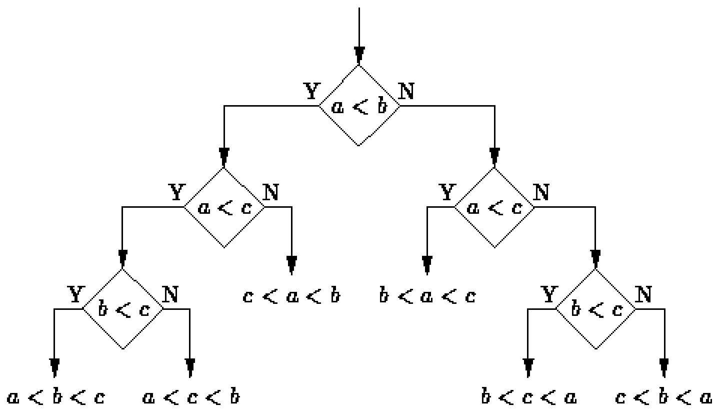
بهترین الگوریتم مرتب سازی که تا حالا بررسی کردیم (مرتب‌سازی ادغامی یا هرمی) در بدترین حالت $\Theta(n\log n)$ عمل مقایسه انجام می‌دهند. سوال پیش می‌آید آیا الگوریتم مرتب‌سازی سریعتری وجود دارد که مرتبه اجرایی بهتری داشته باشد؟ برای مثال $O(n\log\log n)$ یا $O(n)$؟ در این قسمت نشان می‌دهیم الگوریتم‌های مرتب‌سازی که بر اساس مقایسه عناصر عمل می‌کنند، زمان اجرایشان نمی‌تواند از $O(n\log n)$ بهتر باشد. برای نشان دادن این موضوع باید ابتدا مقصودمان را از الگوریتم مرتب سازی بر اساس مقایسه روشن کنیم.

تعریف: یک الگوریتم مرتب سازی بر اساس مقایسه، آرایه A با n عنصر متمایز $a_1,\dots,a_n$ را تنها با تکیه بر مقایسه عناصر مرتب می‌کند و از اطلاعاتی دیگر در مورد محتوای عناصر استفاده نمی‌کند. هر مقایسه $a_i > a_j$? جواب آری یا نه دارد و یک دستورالعمل به حساب می‌آید. در انتها الگوریتم جایگشتی از $a_1,\dots,a_n$ را تولید می‌کند که عناصر آن از کوچک به بزرگ مرتب شده‌اند.

همه الگوریتمهای مرتب‌سازی که در این درس مورد مطالعه قرار گرفته‌اند (مثل حبابی، درجی، ادغامی، هرمی) از نوع مرتب سازی بر اساس مقایسه هستند. الگوریتمهایی مرتب سازی دیگری مثل (مرتب سازی سطلی، مبنایی) وجود دارند که بر اساس محتوای عناصر عمل مرتب‌سازی را انجام می‌دهند.

قضیه: هر الگوریتم مرتب سازی بر اساس مقایسه برای مرتب سازی n عنصر باید حداقل $\Omega(n \log n)$ مقایسه انجام دهد.

اثبات : فرض کنید S یک الگوریتم مرتب‌سازی بر اساس مقایسه باشد. الگوریتم S در طول اجرایش یک سری مقایسه انجام می‌دهد و در نهایت جایگشتی از $a_1, \ldots, a_n$ را به عنوان خروجی تولید می‌کند. مقایسه‌های محتملی که الگوریتم S انجام می‌دهد را می‌توان با یک درخت دودویی T_S نمایش داد. در ریشه درخت اولین مقایسه S قرار می‌گیرد. بسته به نتیجه این مقایسه، یک مقایسه دیگر انجام می‌شود و باز بسته به نتیجه آن مقایسه یک مقایسه دیگر انجام می‌شود. هر مقایسه دو فرزند دارد که در واقع روند کار الگوریتم را در صورت بدست آمدن هر نتیجه نشان می‌دهد. شکل زیر درخت مقایسه یک الگوریتم مرتب‌سازی برای مرتب کردن ورودی (a, b, c) را نشان می‌دهد.



هر برگ درخت مقایسه T_S متناظر با یک جایگشت از ورودی می‌باشد که در نتیجه انجام یک سری مقایسه از ریشه تا آن برگ بدست آمده است. درخت مقایسه T_S باید $n!$ برگ داشته باشد چون همه جایگشتهای ممکن باید قابل تولید باشند (برای هر جایگشت، ورودی را می‌توان طوری انتخاب کرد که آن جایگشت تولید شود). حداقل عمق یک درخت دودویی با $n!$ برگ برابر با $\log(n!)$ است. پس حتما برگی در T_S وجود دارد که از ریشه $\log n!$ فاصله دارد. در نتیجه جایگشتی وجود دارد که برای تولید آن، الگوریتم S به تعداد $\Omega(\log n!) = \Omega(n \log n)$ مقایسه انجام می‌دهد. این قضیه را اثبات می‌کند. $\square$

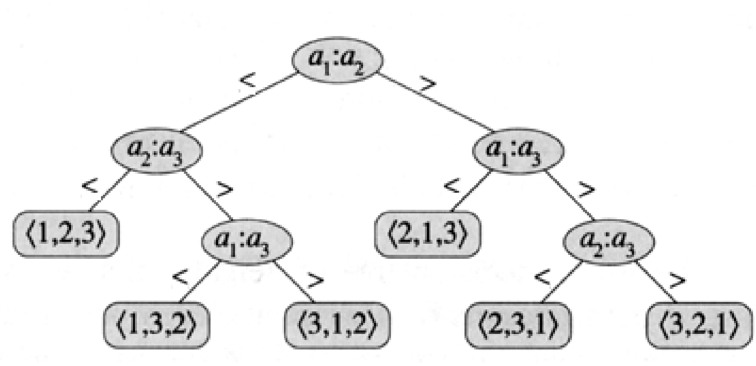


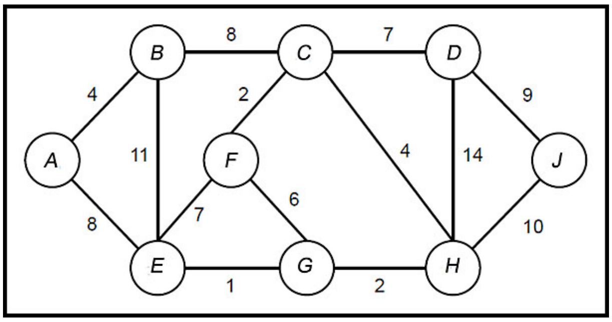
Figure ۱: درخت مقایسه مرتب‌سازی درجی برای مرتب‌سازی a_1, a_2, a_3. هر برگ جایگشتی از ورودی را نشان می‌دهد.

۳ کاربرد ساختار داده هرم بیشینه: الگوریتم پریم

مسئله درخت فراگیر کمینه گراف ساده و همبند $G = (V, E)$ داده شده است. یک تابع هزینه $c : E \to \mathbb{R}$ هم داده شده که هزینه هر یال را مشخص میکند. دقت کنید اینجا هزینه یال میتواند منفی هم باشد. میخواهیم مجموعه ای از یالها $T \subseteq E$ را انتخاب کنیم بطوریکه یالهای انتخابی تشکیل یک درخت را بدهند و مجموع هزینه یالهای انتخابی مینیمم باشد. یعنی درختی فراگیر و کم هزینه تر از آن وجود نداشته باشد.

تعریف: به مجموعه ای از یالها که تشکیل یک درخت بدهد و همه رئوس گراف را در بر بگیرد یک درخت فراگیر گفته میشود.

مسئله به عبارت دیگر این است که در گراف داده شده میخواهیم یک درخت فراگیر با کمترین هزینه پیدا کنیم.



۱.۳ الگوریتم پریم

الگوریتم پریم برای مسئله درخت فراگیر کمینه توسط رابرت پریم ریاضیدان آمریکایی پیشنهاد شده است. ابتدا یک راس دلخواه $s \in V$ انتخاب میشود و مجموعه $S = \{s\}$ مقداردهی میشود. الگوریتم در هر مرحله یک راس $x \in V \setminus S$ را به جمع رئوس S اضافه میکند. راسی انتخاب میشود که با کمترین هزینه به S وصل شود. یالی که راس مورد نظر را به S وصل میکند به درخت در حال رشد T که در ابتدا تهی است اضافه میشود. کار ادامه پیدا میکند تا زمانیکه S همه گراف را در بر بگیرد.

۱. راس دلخواه $s \in V$ را انتخاب کن.

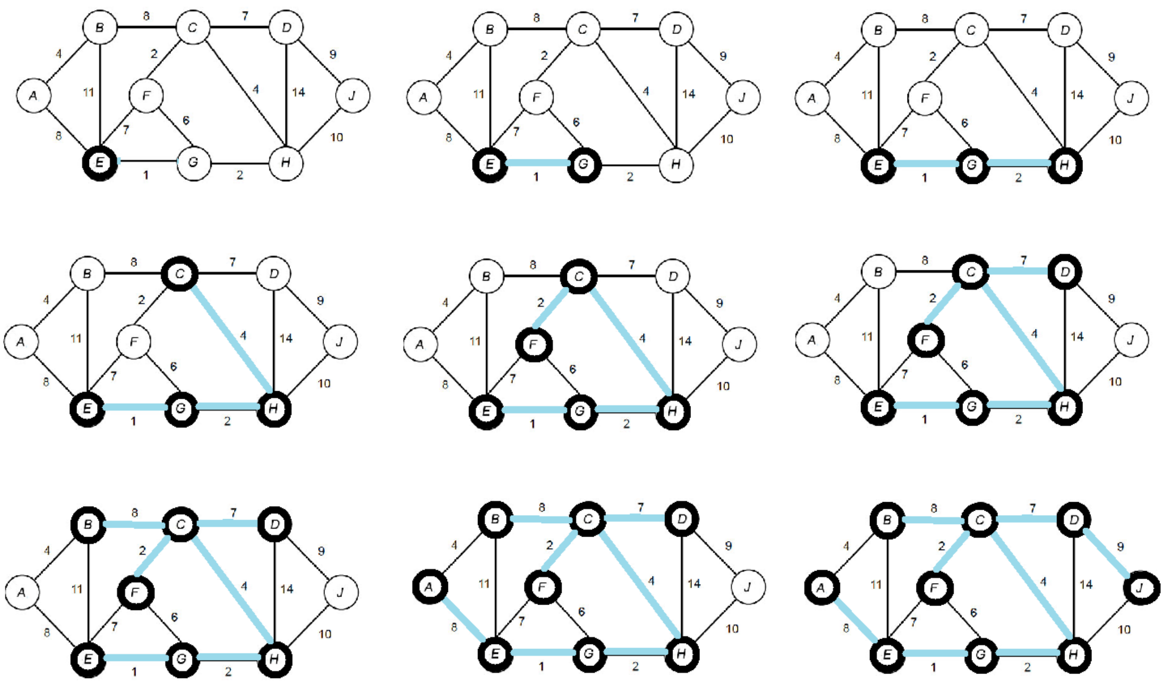
۲. قرار بده $S = \{s\}$.

۳. $T \leftarrow \emptyset$

۴. تا زمانیکه $S \neq V$ کارهای زیر را انجام بده:

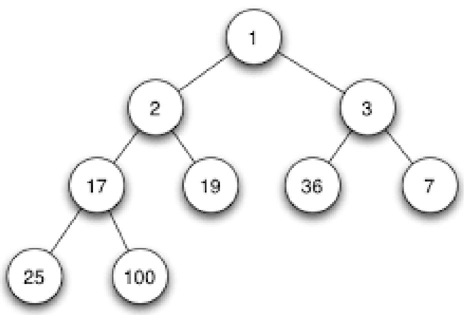
- یال (y, x) را پیدا کن که یک سر در S و سر دیگر در $V \setminus S$ داشته باشد و کمترین وزن را داشته باشد.
- یال انتخابی (y, x) در قدم قبلی را به T اضافه کن. همچنین راس x را به S اضافه کن.

یک نمونه از اجرای الگوریتم پریم در شکل زیر نشان داده شده است.



۲.۳ هرم کمینه برای پیاده سازی الگوریتم پریم

هرم کمینه مانند هرم بیشینه است با این تفاوت که عنصری که در ریشه قرار میگیرد باید کلیدش کمتر یا مساوی کلید عناصر واقع در زیردرختهایش باشد. یک نمونه هرم کمینه در زیر نشان داده شده است.



همانطور که در توضیح الگوریتم پریم دیدیم، هر بار یک یال (x, y) از برش $(S, V \setminus S)$ که کمترین وزن را دارد انتخاب میشود و به درخت فعلی T اضافه میشود. سپس راس جدید y به مجموعه S اضافه میشود. بدین ترتیب برش $(S, V \setminus S)$ تغییر میکند. با اضافه شدن y به S یک سری یال از برش حذف شده و یک سری یال دیگر به برش اضافه میشوند.

برای نگهداری یالهای برش، میتوانیم از صف اولویت (هرم کمینه) استفاده کنیم. عناصر هرم یالها هستند و کلیدها وزن یالها میباشد. یال با کمترین وزن در ریشه هرم قرار میگیرد و لذا پیدا کردن آن سریع خواهد بود. علاوه بر این، حذف و اضافه کردن یال به هرم میتواند در زمان $\log m$ انجام شود زیرا حداکثر m یال در هرم قرار میگیرد و ارتفاع هرم حداکثر $\log m$ خواهد بود.

ما اینجا یک ایده مقداری متفاوت با آنچه گفته شد را با استفاده از هرم کمینه پیاده سازی میکنیم. به جای اینکه یالها به هرم اضافه یا حذف شوند، همه یالهای گراف در ابتدای کار به هرم اضافه میشوند و هرگز از هرم حذف نمیشوند! بجای آن فقط وزن یالها تغییر میکند. مقدار کلید همه یالها را در شروع کار بینهایت ∞ فرض میشود. سپس هر موقع در طول اجرای الگوریتم یال (x, y) در برش $(S, V \setminus S)$ قرار گرفت مقدار کلید یال مربوطه از بینهایت به $w(x, y)$ که وزن واقعی یال است کاهش پیدا میکند. هر موقع یال (x, y) از برش $(S, V \setminus S)$ خارج شد، مقدار کلید یال مربوطه به بینهایت برمیگردد (این مثل حذف یال عمل میکند گرچه در واقع یال از هرم حذف نمیشود.) پس تنها عمل مربوط به هرم که اینجا مورد نیاز است عمل تغییر کلید است که آن هم در زمان متناسب با ارتفاع هرم قابل انجام است.

برای پیاده سازی، از چهار ساختار داده (سه آرایه و یک هرم) استفاده میکنیم.

- آرایه $EDGES$ یالها را به یک ترتیب دلخواه که در اول کار مشخص میشود نگهداری میکند.

  هر عنصر آرایه $EDGES$ دو مولفه دارد. $EDGES[i][0]$ اسم یال مربوطه را نگهداری میکند. برای مثال $EDGES[i][0] = (A, B)$ یعنی اینکه یال (A, B) در اندیس i آرایه ذخیره شده است.

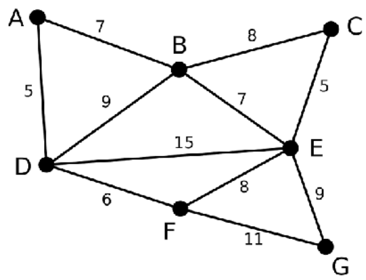
  در مولفه دوم، محل نگهداری یال مربوطه در هرم ذخیره میشود. اینجا $EDGES[i][1] = j$ یعنی اینکه یالی که در اندیس i ذخیره شده است، در اندیس j هرم قرار گرفته است.

- هرم $HEAP$ که با یک آرایه با طول $m + 1$ پیاده سازی شده است، برای نگهداری یالها استفاده میشود. هر عنصر هرم مربوط به یک یال گراف است. در بخش کلید وزن یال ذخیره میشود و در بخش داده مشخصات یال به اضافه اندیس آن در آرایه $EDGES$ نگهداری میشود.

- گراف ورودی بصورت آرایه ای از لیستها ذخیره میشود. برای هر راس گراف $v \in V$ لیست یالهای واقع روی راس v نگهداری میشود. ترتیب یالها در لیست دلخواه است. برای هر یال اندیس یال مربوطه در آرایه $EDGES$ هم نگهداری میشود.

- برای مشخص کردن مجموعه S آرایه S به طول n نگهداری میشود. $S[i] = True$ به این معنی است که راس i در مجموعه S قرار گرفته است. در شروع کار برای همه i ها داریم $S[i] = False$.

یک نمونه از اجرای الگوریتم در سری شکلهای زیر نمایش داده شده است.



INPUT GRAPH

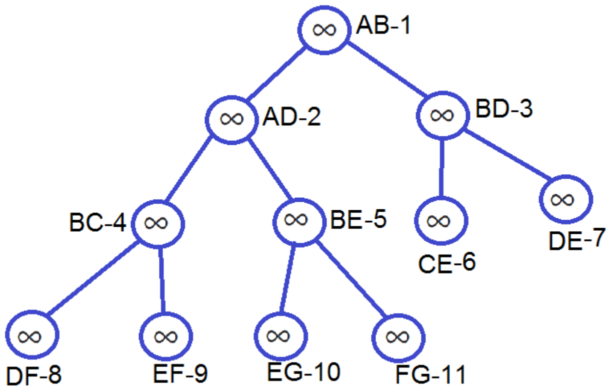
AB	1
AD	2
BD	3
BC	4
BE	5
CE	6
DE	7
DF	8
EF	9
EG	10
FG	11

EDGES

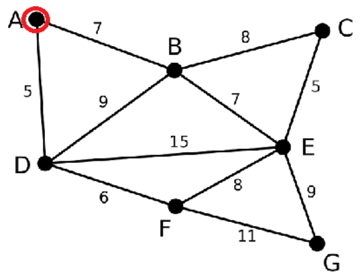
A		AB-1, AD-2
B		BA-1, BC-4, BD-3, BE-5
C		CB-4, CE-6
D		DA-2, DB-3, DE-7, DF-8
E		EB-5, EC-6, ED-7, EF-9, EG-10
F		FE-9, FD-8, FG-11
G		GE-10, GF-11

S

LIST OF EDGES



MIN HEAP



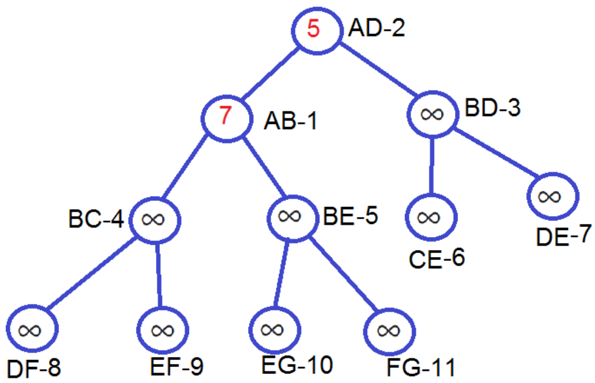
INPUT GRAPH

AB	2
AD	1
BD	3
BC	4
BE	5
CE	6
DE	7
DF	8
EF	9
EG	10
FG	11

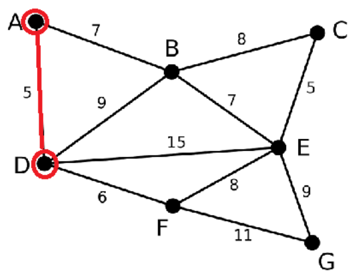
EDGES

A		AB-1, AD-2
B		BA-1, BC-4, BD-3, BE-5
C		CB-4, CE-6
D		DA-2, DB-3, DE-7, DF-8
E		EB-5, EC-6, ED-7, EF-9, EG-10
F		FE-9, FD-8, FG-11
G		GE-10, GF-11

S LIST OF EDGES



MIN HEAP



INPUT GRAPH

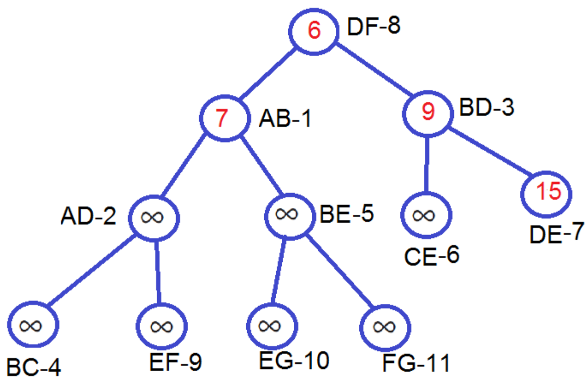
AB	2
AD	4
BD	3
BC	8
BE	5
CE	6
DE	7
DF	1
EF	9
EG	10
FG	11

EDGES

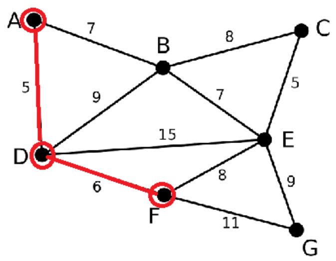
A		AB-1, AD-2
B		BA-1, BC-4, BD-3, BE-5
C		CB-4, CE-6
D		DA-2, DB-3, DE-7, DF-8
E		EB-5, EC-6, ED-7, EF-9, EG-10
F		FE-9, FD-8, FG-11
G		GE-10, GF-11

S

LIST OF EDGES



MIN HEAP



INPUT GRAPH

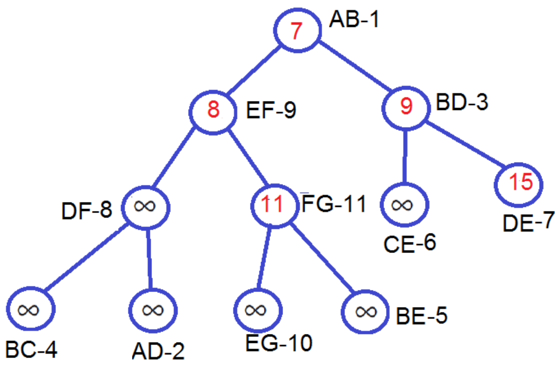
AB	1
AD	9
BD	3
BC	8
BE	11
CE	6
DE	7
DF	4
EF	2
EG	10
FG	5

EDGES

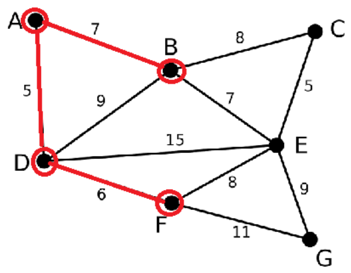
S		
A		AB-1, AD-2
B		BA-1, BC-4, BD-3, BE-5
C		CB-4, CE-6
D		DA-2, DB-3, DE-7, DF-8
E		EB-5, EC-6, ED-7, EF-9, EG-10
F		FE-9, FD-8, FG-11
G		GE-10, GF-11

S

LIST OF EDGES



MIN HEAP



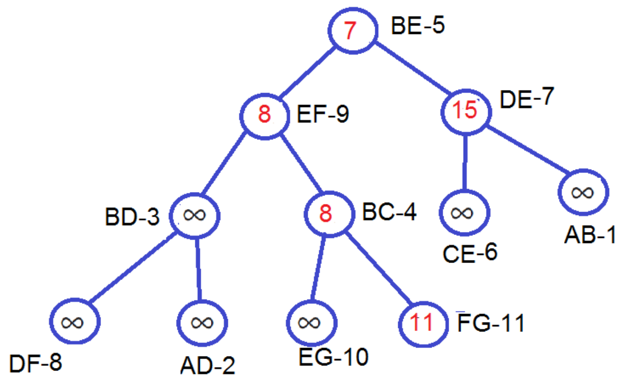
INPUT GRAPH

AB	7
AD	9
BD	4
BC	5
BE	1
CE	6
DE	3
DF	8
EF	2
EG	10
FG	11

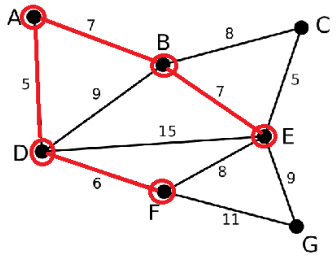
EDGES

A		AB-1, AD-2
B		BA-1, BC-4, BD-3, BE-5
C		CB-4, CE-6
D		DA-2, DB-3, DE-7, DF-8
E		EB-5, EC-6, ED-7, EF-9, EG-10
F		FE-9, FD-8, FG-11
G		GE-10, GF-11

S **LIST OF EDGES**



MIN HEAP



INPUT GRAPH

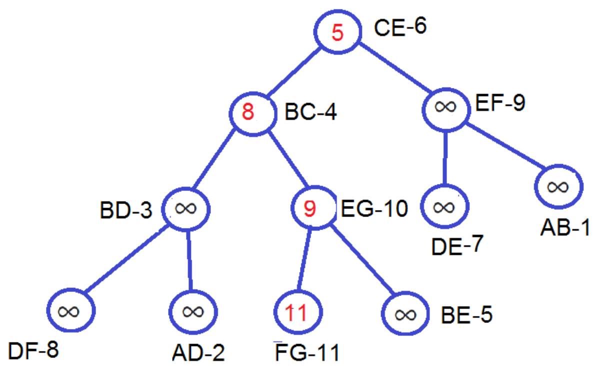
AB	7
AD	9
BD	4
BC	2
BE	11
CE	1
DE	6
DF	8
EF	3
EG	5
FG	10

EDGES

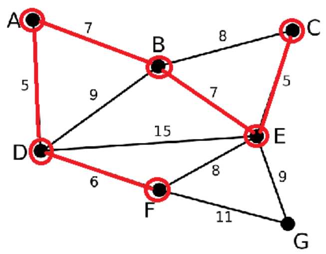
A		AB-1, AD-2
B		BA-1, BC-4, BD-3, BE-5
C		CB-4, CE-6
D		DA-2, DB-3, DE-7, DF-8
E		EB-5, EC-6, ED-7, EF-9, EG-10
F		FE-9, FD-8, FG-11
G		GE-10, GF-11

S

LIST OF EDGES



MIN HEAP



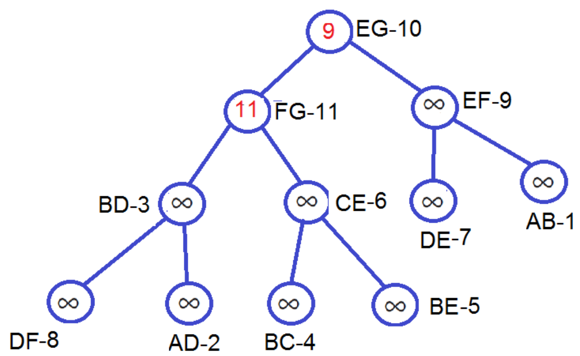
INPUT GRAPH

AB	7
AD	9
BD	4
BC	10
BE	11
CE	5
DE	6
DF	8
EF	3
EG	1
FG	2

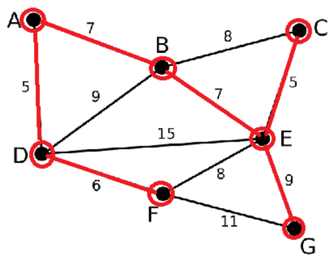
EDGES

A		AB-1, AD-2
B		BA-1, BC-4, BD-3, BE-5
C		CB-4, CE-6
D		DA-2, DB-3, DE-7, DF-8
E		EB-5, EC-6, ED-7, EF-9, EG-10
F		FE-9, FD-8, FG-11
G		GE-10, GF-11

S LIST OF EDGES



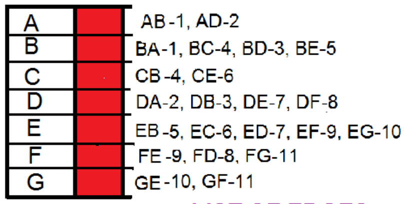
MIN HEAP



INPUT GRAPH

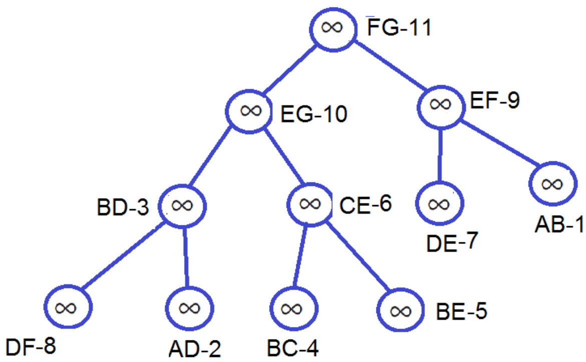
AB	7
AD	9
BD	4
BC	10
BE	11
CE	5
DE	6
DF	8
EF	3
EG	2
FG	1

EDGES



S

LIST OF EDGES



MIN HEAP