HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

ĐỒ THỊ (PHẦN 2)

ONE LOVE. ONE FUTURE.

NỘI DUNG

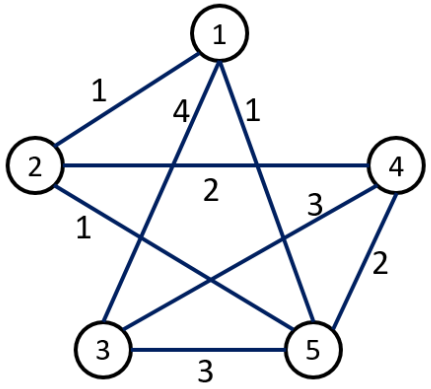
- Thuật toán Kruskal và cấu trúc Disjoint Set giải bài toán cây khung nhỏ nhất
- Thuật toán Dijkstra và cấu trúc Priority Queue giải bài toán đường đi ngắn nhất

THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

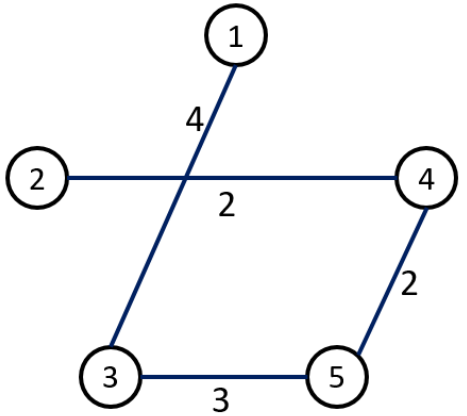
- Bài toán cây khung nhỏ nhất của đồ thị
  - Cho đồ thị vô hướng liên thông $G = (V, E)$ trong đó $V = \{1, 2, \ldots, n\}$ là tập đỉnh và E là tập cạnh
    - $c(u,v)$ là trọng số cạnh (u,v), với mọi $(u,v) \in E$
  - Cây $T = (V, F)$ trong đó $F \subseteq E$ được gọi là một cây khung của G
  - Yêu cầu: tìm cây khung của G với trọng số nhỏ nhất

THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

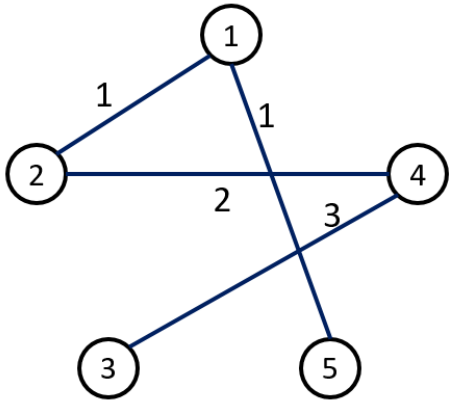
- Bài toán cây khung nhỏ nhất của đồ thị
  - Cho đồ thị vô hướng liên thông $G = (V, E)$ trong đó V là tập đỉnh và E là tập cạnh
    - $c(u,v)$ là trọng số cạnh (u,v), với mọi $(u,v) \in E$
  - Cây $T = (V, F)$ trong đó $F \subseteq E$ được gọi là một cây khung của G
  - Yêu cầu: tìm cây khung của G với trọng số nhỏ nhất



Đồ thị G



Cây khung T_1, trọng số 11



Cây khung T_2, trọng số 7

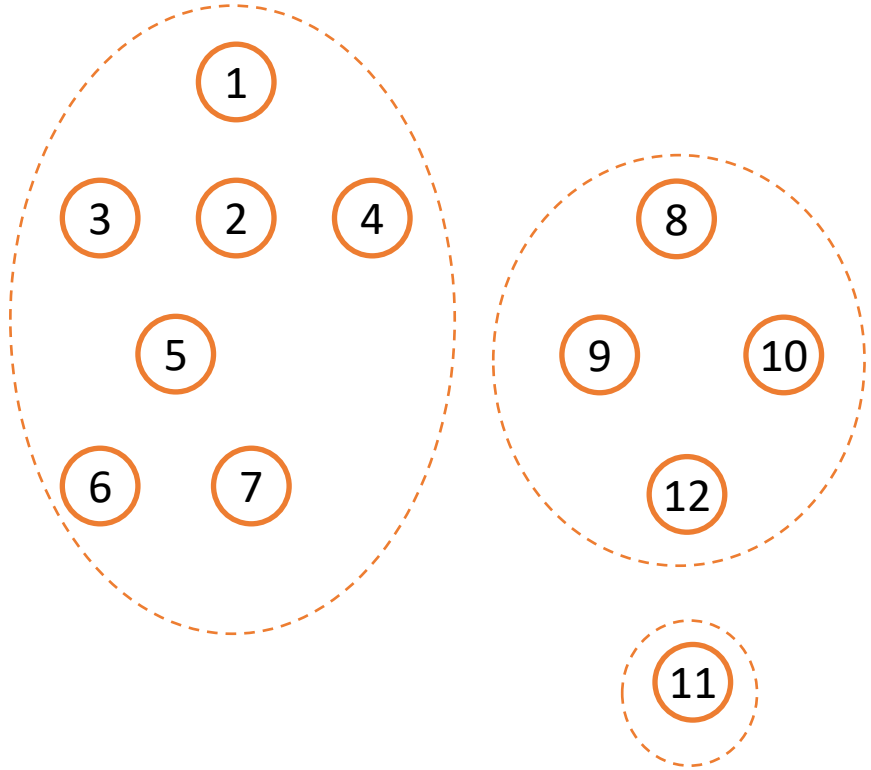
THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

- Kruskal là một thuật toán tham lam:
  - Ban đầu, lời giải chỉ là tập các đỉnh của G
  - Mỗi bước lặp, ta chọn ra 1 cạnh có trọng số nhỏ nhất để bổ sung vào lời giải với điều kiện không được tạo ra chu trình.
  - Quá trình lặp sẽ kết thúc khi tất cả các đỉnh của đồ thị được kết nối liên thông với nhau

```
Kruskal(G = (V, E)) {
  T = {};
  L = sort E in a non-decreasing order of weight;
  for (u,v) in L do {
    if T ∪ {(u,v)} does not create a cycle then {
       T = T ∪ {(u,v)};
       if |T| = |V| - 1 then break;
    }
  }
  if |T| < |V| - 1 then
    return NULL;
  else
    return T;
}
```

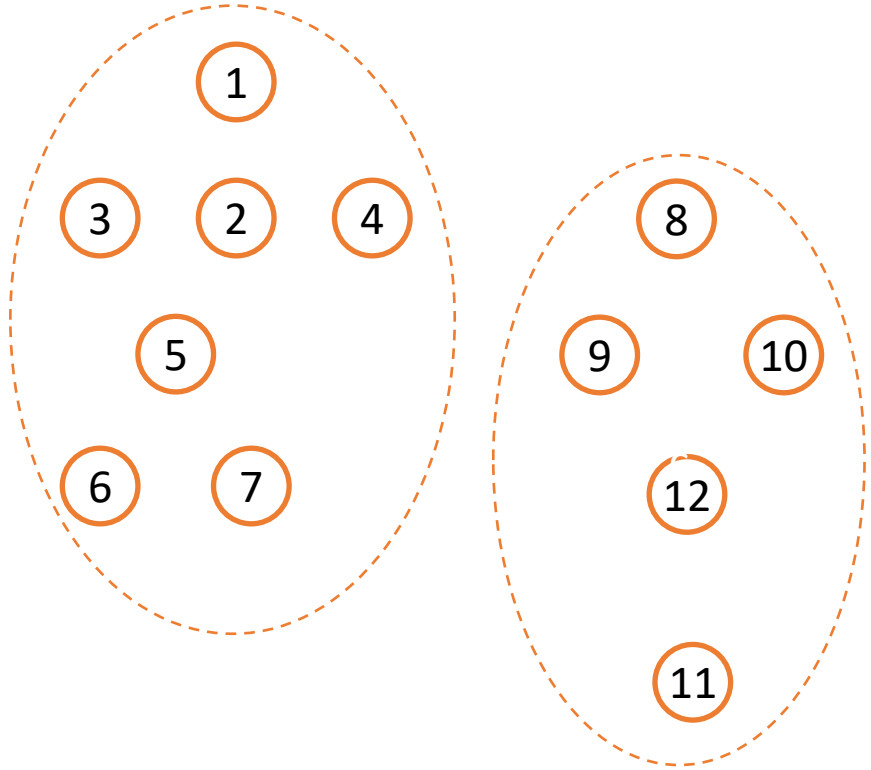
THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

- Disjoint Set: Là cấu trúc dữ liệu biểu diễn các tập không giao nhau với 2 thao tác chính
  - Find(x): trả về định danh của tập chứa x
  - Unify(r1, r2): Hợp nhất 2 tập hợp định danh là r1 và r2 làm một



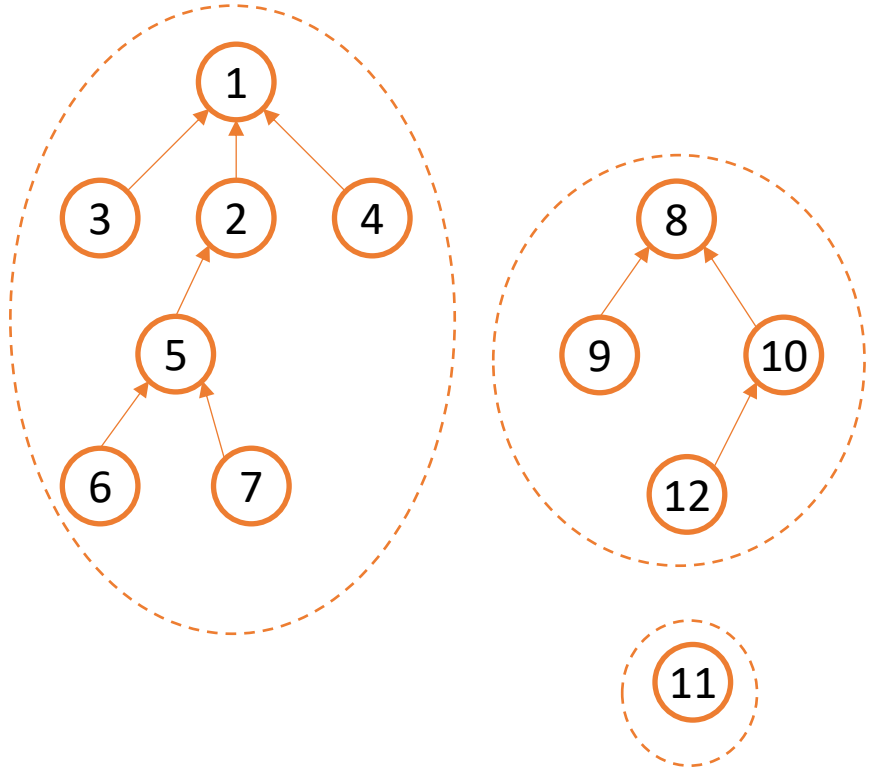
THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

- Disjoint Set: Là cấu trúc dữ liệu biểu diễn các tập không giao nhau với 2 thao tác chính
  - Find(x): trả về định danh của tập chứa x
  - Unify(*u*, *v*): Hợp nhất 2 tập hợp định danh là *u* và *v* làm một



THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

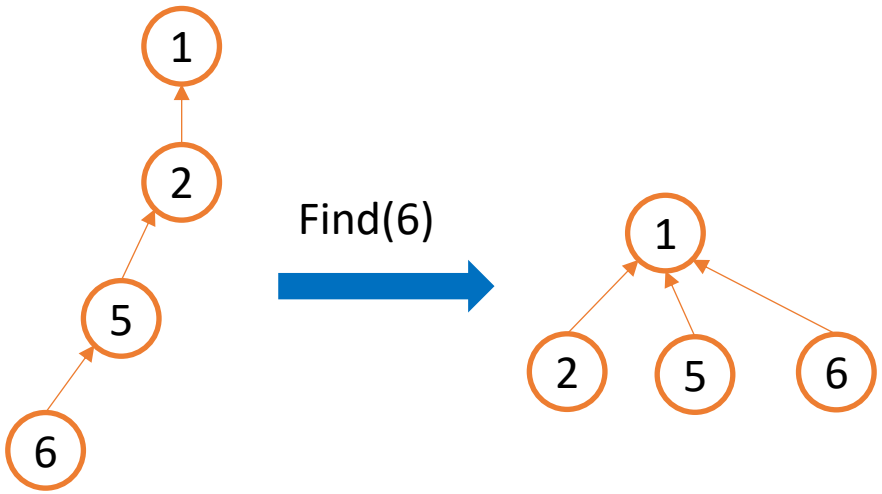
- Disjoint Set: Là cấu trúc dữ liệu biểu diễn các tập không giao nhau với 2 thao tác chính
  - Find(x): trả về định danh của tập chứa x
  - Unify(*u*, *v*): Hợp nhất 2 tập hợp định danh là *u* và *v* làm một
- Mỗi tập được biểu diễn bởi cây có gốc
  - Mỗi nút của cây là một phần tử
  - Mỗi nút *x* có 1 nút cha duy nhất *p*[*x*] (cha của nút gốc là chính nó)
  - Nút gốc là định danh của tập



THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

- Thao tác Find(*x*):
  - Xuất phát từ *x*, liên tiếp truy cập đến nút cha để tìm đến nút gốc
  - Độ phức tạp tỉ lệ với độ dài của đường đi từ *x* đến gốc
  - Path Compression: trong quá trình đi từ *x* đến gốc, các nút trên đường đi sẽ được gắn như là nút con trực tiếp của nút gốc để giảm độ dài đường đi từ các nút này đến gốc trong các bước lặp sau

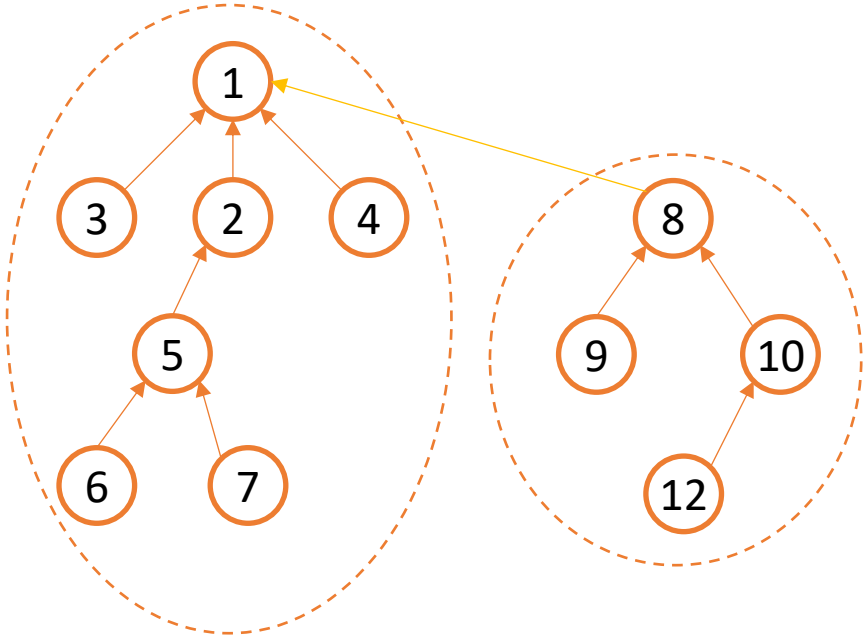
```
Find(x){
    if(x != p[x]) p[x] = Find(p[x]);
    return p[x];
}
```



THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

- Thao tác Unify(*u*, *v*):
  - Cho *u* là con của *v* hoặc *v* là con của *u* tùy thuộc độ cao của nút nào nhỏ hơn
  - Mỗi nút, duy trì *r*[*x*] thể hiện độ cao của nút *x*

```
Unify(x, y){
    if(r[x] > r[y]) p[y] = x;
    else{
        p[x] = y;
        if(r[x] == r[y]) r[y] = r[y] + 1;
    }
}
```



THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

```
makeSet(x){
    p[x] = x; r[x] = 0;
}
Find(x){
    if(x != p[x]) p[x] = Find(p[x]);
    return p[x];
}
Unify(x, y){
    if(r[x] > r[y]) p[y] = x;
    else{
        p[x] = y;
        if(r[x] == r[y]) r[y] = r[y] + 1;
    }
}
```

```
KruskalWithDisjointSset(G = (V, E)){
  T = {};
  for v in V do makeSet(v);
  L = sort E in a non-decreasing order of weight;
  for (u,v) in L do {
     ru = Find(u);
     rv = Find(v);
     if ru ≠ rv then {
        Unify(ru, rv);
        T = T ∪ {(u,v)};
        if |T| == |V| - 1 then break;
     }
  }
  if |T|  < |V| - 1 then return {};
  return T;
}
```

THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

- Minh họa với ngôn ngữ C
- Dữ liệu
  - Dòng 1: ghi 2 số nguyên dương n và m tương ứng là số đỉnh và số cạnh của G ($1 \le n, m \le 10^5$)
  - Dòng *i* (*i* = 1, 2, …, m): ghi 3 số nguyên dương *u*, *v* and *c* trong đó *c* là trọng số cạnh (*u*,*v*)
- Kết quả
  - Ghi ra trọng số của cây khung nhỏ nhất tìm được

stdin	stdout
5 8 1 2 1 1 3 4 1 5 1 2 4 2 2 5 1 3 4 3 3 5 3 4 5 2	7

THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

```
#include <stdio.h>
#define MAX 100001
// data structure for input graph
int N, M;
int u[MAX];
int v[MAX];
int c[MAX];
int ET[MAX];
int nET;

// data structure for disjoint-set
int r[MAX];// r[v] is rank of set v
int p[MAX];// p[v] is parent of v
long long rs;
```

```
void unify(int x, int y){
    if(r[x] > r[y]) p[y] = x;
    else{
        p[x] = y;
        if(r[x] == r[y]) r[y] = r[y] + 1;
    }
}
void makeSet(int x){
    p[x] = x;  r[x] = 0;
}
int findSet(int x){
    if(x != p[x])  p[x] = findSet(p[x]);
    return p[x];
}
```

THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

```
void swapEdge(int i, int j){
  int tmp = c[i]; c[i] = c[j]; c[j] = tmp;
  tmp = u[i]; u[i] = u[j]; u[j] = tmp;
  tmp = v[i]; v[i] = v[j]; v[j] = tmp;
}
int partition(int L, int R, int index){
  int pivot = c[index]; swapEdge(index,R);
  int storeIndex = L;
  for(int i = L; i <= R-1; i++){
    if(c[i] < pivot){
      swapEdge(storeIndex,i); storeIndex++;
    }
  }
  swapEdge(storeIndex,R); return storeIndex;
}
```

```
void quickSort(int L, int R){
    if(L < R){
        int index = (L+R)/2;
        index = partition(L,R,index);
        if(L < index) quickSort(L,index-1);
        if(index < R) quickSort(index+1,R);
    }
}
void sort(){
    quickSort(0,M-1);
}
```

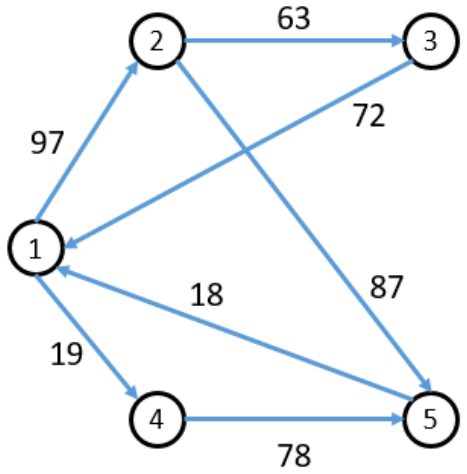
THUẬT TOÁN KRUSKAL VÀ CẤU TRÚC DISJOINT SET

```
void kruskal(){
    for(int x = 1; x <= N; x++)  makeSet(x);
    sort(); rs = 0; int nET = 0;
    for(int i = 0;i < M; i++){
        int ru = findSet(u[i]);
        int rv = findSet(v[i]);
        if(ru != rv){
            unify(ru,rv);
            nET++;   rs += c[i];
            if(nET == N-1) break;
        }
    }
    printf("%lld",rs);
}
```

```
void input(){
    scanf("%d%d",&N,&M);
    for(int i = 0; i < M; i++){
        scanf("%d%d%d",&u[i],&v[i],&c[i]);
    }
}
int main(){
    input();
    kruskal();
}
```

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Bài toán đường đi ngắn nhất giữa 2 đỉnh trên đồ thị trọng số không âm
  - Cho đồ thị có hướng liên thông $G = (V, E)$ trong đó $V = \{1, 2, \ldots, n\}$ là tập đỉnh và E là tập cung
    - $c(u,v)$ là trọng số không âm của cung (u,v), với mọi $(u,v) \in E$
  - Yêu cầu: Cho 2 đỉnh s và t thuộc G, hãy tìm đường đi có tổng trọng số nhỏ nhất trên G



Đồ thị G: đường đi ngắn nhất từ 1 đến 5 là 1 - 4 - 5 với độ dài bằng 19 + 78 = 97

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Cấu trúc dữ liệu biểu diễn đồ thị G sử dụng danh sách kề
  - $A[u]$ là tập các cung e đi ra khỏi đỉnh u, mọi $u \in V$.
  - Với mỗi cung e đi ra khỏi đỉnh u thì $e.id$ là đỉnh còn lại của e và $e.w$ là trọng số cung
    - Ví dụ: cung $e = (u, v)$ là một cung đi ra khỏi u có trọng số 10, khi đó $e.id = v$ và $e.w = 10$

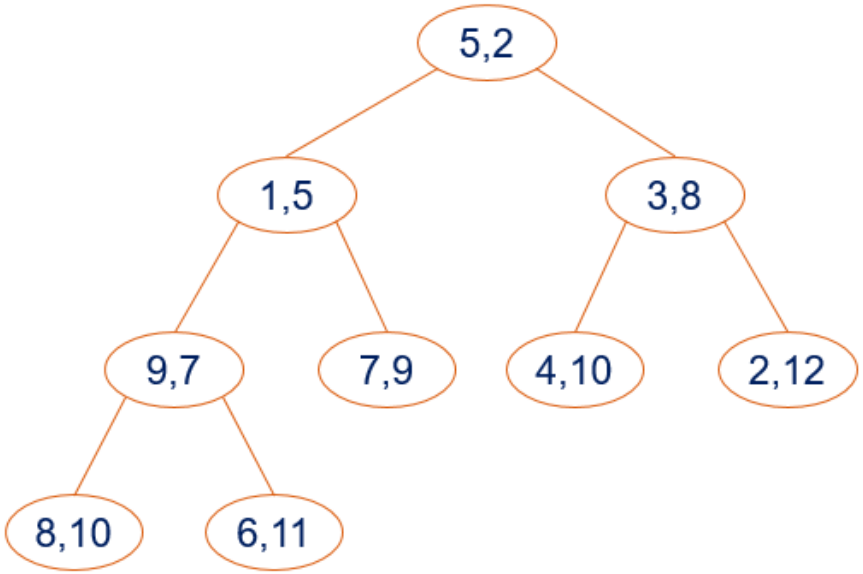
THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Thuật toán Dijkstra
  - Với mỗi đỉnh v của G, ta duy trì một đường đi cận trên $P[v]$ của đường đi ngắn nhất từ s đến v. Ký hiệu $d[v]$ là độ dài của $P[v]$. Đường đi cận trên này sẽ được làm tốt dần lên (độ dài giảm dần) qua các bước lặp.
  - Nếu tồn tại một đỉnh u sao cho $d[v] > d[u] + c(u,v)$ thì ta làm tốt đường đi cận trên $P[v]$ bằng đường đi $P[u]$ nối thêm cung (u,v), đồng thời cập nhật $d[v] = d[u] + c(u,v)$

```
Dijkstra(G = (V, E)){
  for v in V do d[v] = ∞;
  d[s] = 0; F = V;
  while F not empty do {
     u = select u from F s.t. d[u] is minimal;
     if u = t then break;
     F = F \ {u};
     for e in A[u] do {
       if d[e.id] > d[u] + e.w then
          d[e.id] = d[u] + e.w;
       }
  }
  return d[t];
}
```

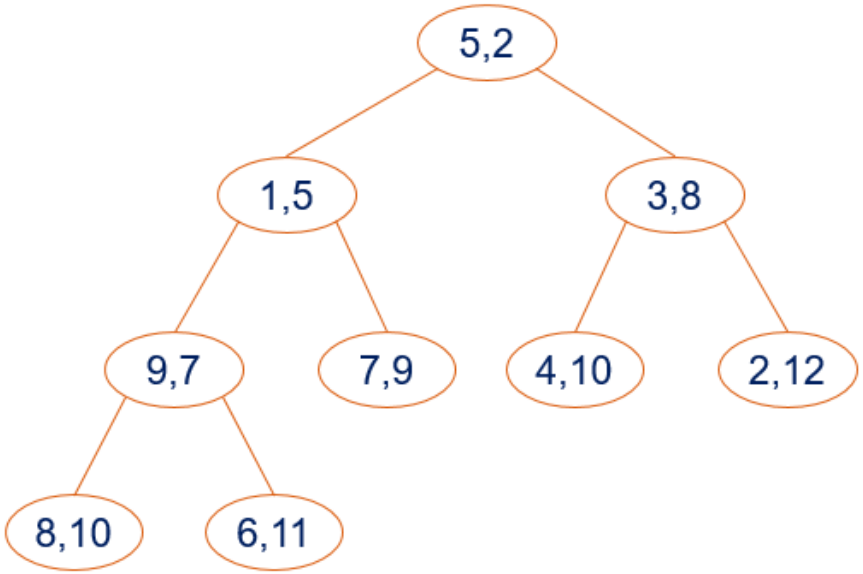
THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Priority Queue (hàng đợi ưu tiên)
  - Cấu trúc dữ liệu lưu trữ các cặp phần tử *v* và khóa của nó *d*[*v*], với 2 thao tác chính:
    - push(*v*, *d*[*v*]): đưa cặp (*v*, *d*[*v*]) vào hàng đợi ưu tiên hoặc cập nhật lại khóa của *v* nếu phần tử (cặp) định danh *v* đã tồn tại
    - deleteMin(): loại bỏ cặp (*v*, *d*[*v*]) có khóa *d*[*v*] nhỏ nhất trong số các cặp trong hàng đợi ưu tiên và trả về phần tử *v*.



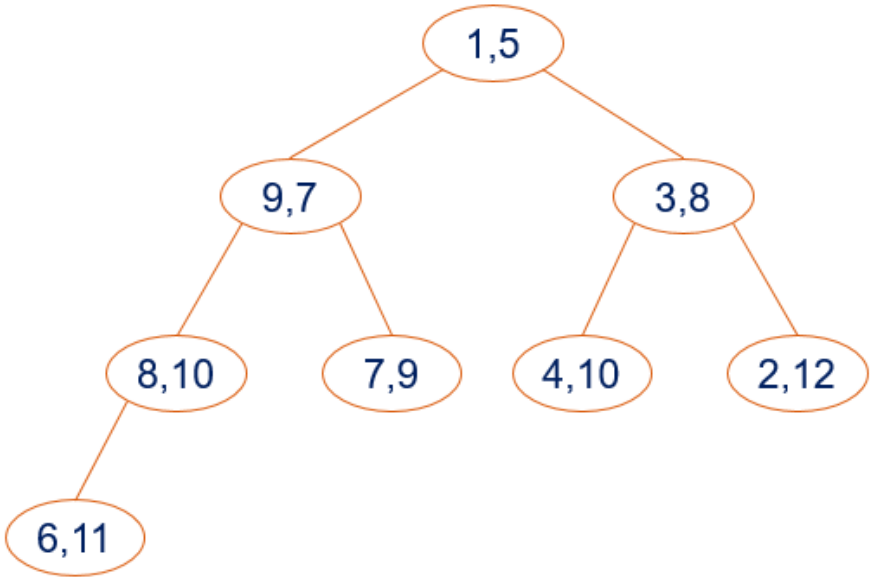
THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Priority Queue (hàng đợi ưu tiên)
  - Cài đặt:
    - Sử dụng mảng với các phần tử được đánh số 0, 1, 2, …
      - Mỗi phần tử chứa 2 thông tin *v* là định danh và *d*[*v*] là khóa của phần tử
    - Mảng được nhìn dưới góc độ một cây nhị phân đầy đủ
      - Phần tử có số thứ tự *i* thì con trái có số thứ tự 2*i*+1 và con phải có số thứ tự 2*i*+2
      - Khóa của một phần tử nhỏ hơn hoặc bằng khóa của 2 phần tử con (nếu có) → cấu trúc Min-Heap



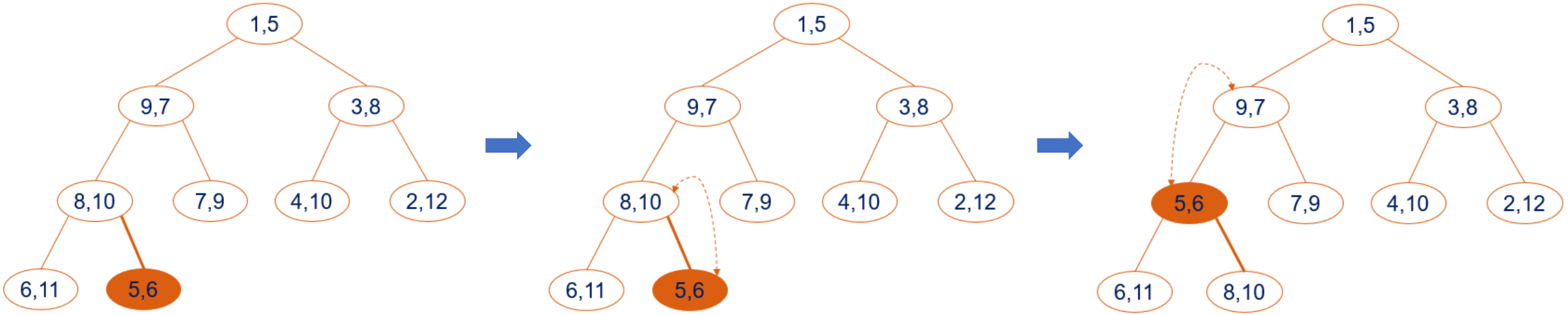
THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Thao tác push(v, $d[v]$)
  - Tạo ra 1 phần tử mới định danh v và khóa $d[v]$
  - Thêm phần tử mới này vào cuối Min-Heap (cuối mảng)
  - Lặp lại việc hoán đổi phần tử này với phần tử cha của nó chừng nào khóa của phần tử này nhỏ hơn khóa của cha (thao tác này gọi là UpHeap).



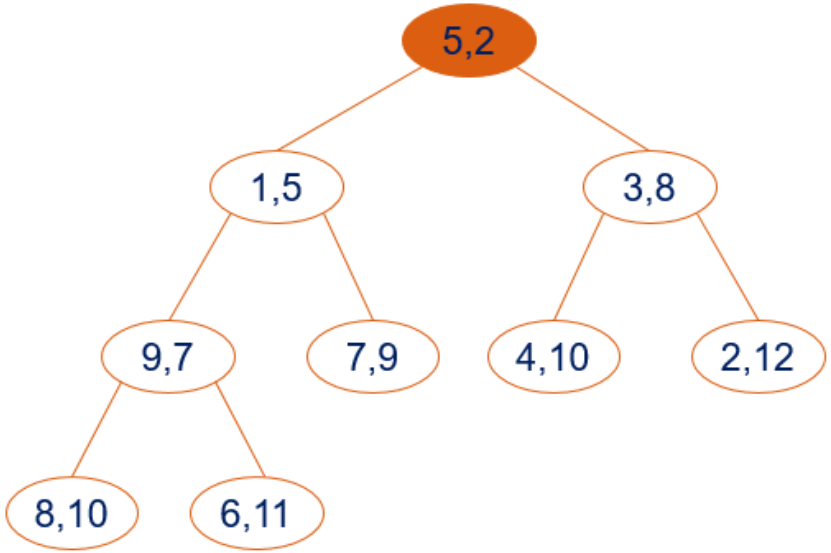
Thực hiện thao tác push(5, 6)

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

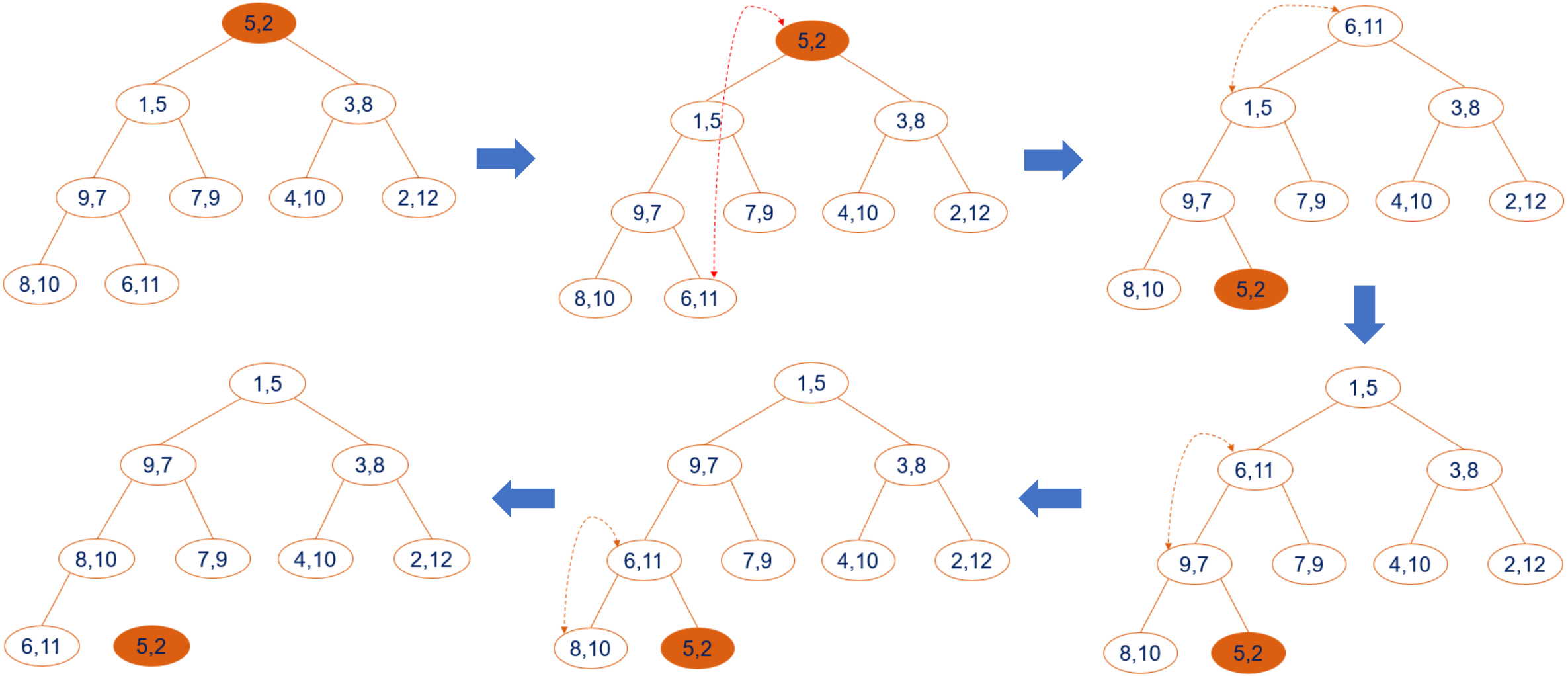


THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Thao tác deleteMin()
  - Hoán đổi phần tử đầu tiên (gốc của Min-Heap) với phần tử cuối cùng của mảng
  - Thực hiện lặp lại việc hoán đổi phần tử hiện tại (xuất phát từ gốc) với phần tử có khóa nhỏ hơn trong số 2 phần tử con (nếu có) chừng nào khóa của phần tử hiện tại còn chưa nhỏ hơn hoặc bằng khóa của các phần tử con (thao tác này còn được gọi là DownHeap)
  - Loại bỏ phần tử cuối cùng của mảng đi



THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE



THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Thuật toán Dijkstra sử dụng hàng đợi ưu tiên
  - Khởi tạo hàng đợi ưu tiên **pq** chứa các đỉnh đã tìm được đường đi cận trên đồng thời chưa tìm được đường đi ngắn nhất
  - Mỗi bước lặp sẽ dùng thao tác deleteMin() để lấy ra đỉnh *u* từ **pq** có *d*[*u*] nhỏ nhất và cập nhật lại *d*[*v*] với mỗi đỉnh *v* kề với *u* nếu d[v] > *d*[*u*] + *c*(*u*,*v*) sau đó push(*v*, *d*[*v*]) vào **pq**

```
Dijkstra(G = (V, A), s, t){
  for v in V do d[v] = +∞;
  pq = initPQ(); pq.push(s, 0);
  while(pq not empty){
    u = pq.deleteMin();
    for e in A[u] do {
       v = e.id; w = e.w;
       if d[v] > d[u] + w then
          pq.push(v,d[u] + w);
    }
  }
  return d[t];
}
```

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

- Minh họa với ngôn ngữ C
- Dữ liệu
  - Dòng 1: ghi 2 số nguyên dương *n* và *m* tương ứng là số đỉnh và số cạnh của G ($1 \le n, m \le 10^5$)
  - Dòng *i* (*i* = 1, 2, …, m): ghi 3 số nguyên dương *u*, *v* and *c* trong đó *c* là trọng số cung (*u*,*v*)
  - Dòng cuối cùng chứa 2 số nguyên dương là đỉnh *s* (đỉnh đầu) và *t* (đỉnh cuối)
- Kết quả
  - Ghi ra trọng số của đường đi ngắn nhất từ *s* đến *t*

stdin	stdout
5 7 2 5 87 1 2 97 4 5 78 3 1 72 1 4 19 2 3 63 5 1 18 1 5	97

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

```
#include <stdio.h>
#include <stdlib.h>
#define N 100001
#define INF 1000000
typedef struct Arc{
  int id;
  int w;
  struct Arc* next;
}Arc;
```

```
int n,m; // number of nodes and arcs of the given graph
int s,t; // source and destination nodes
Arc* A[N]; // A[v] is the pointer to the first item of the adjacent arcs
           // of node v

// priority queue data structure (implemented using BINARY HEAP)
int d[N]; // d[v] is the upper bound of the length of the shortest path
          // from s to v (key)
int node[N]; // node[i] the i^th element in the HEAP
int idx[N]; // idx[v] is the index of v in the HEAP (idx[node[i]] = i)
int sH; // size of the HEAP
```

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

```
void swap(int i, int j){
  int tmp = node[i]; node[i] = node[j];
  node[j] = tmp;
  idx[node[i]] = i; idx[node[j]] = j;
}

void upHeap(int i){
  if(i == 0) return;
  while(i > 0){
    int pi = (i-1)/2;
    if(d[node[i]] < d[node[pi]]) swap(i,pi);
    else break;
    i = pi;
  }
}
```

```
int inHeap(int v){
    return idx[v] >= 0;
}
void downHeap(int i){
  int L = 2*i+1; int R = 2*i+2;
  int maxIdx = i;
  if(L < sH && d[node[L]] < d[node[maxIdx]])
    maxIdx = L;
  if(R < sH && d[node[R]] < d[node[maxIdx]])
    maxIdx = R;
  if(maxIdx != i){
    swap(i,maxIdx); downHeap(maxIdx);
  }
}
```

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

```
void updateKey(int v, int k){
  if(d[v] > k){ d[v] = k; upHeap(idx[v]); }
  else{  d[v] = k; downHeap(idx[v]);  }
}

void pushPQ(int v, int k){
  if(!inHeap(v)){
    d[v] = k;      node[sH] = v;
    idx[node[sH]] = sH;     upHeap(sH);
         sH++;
  }else updateKey(v,k);
}
```

```
int pqEmpty(){
         return sH <= 0;
}

int deleteMin(){
    int sel_node = node[0];
    swap(0,sH-1);  sH--;  downHeap(0);
    return sel_node;
}
```

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

```
Arc* makeArc(int id, int w){
  Arc* a = (Arc*)malloc(sizeof(Arc));
  a->id = id; a->w = w; a->next = NULL; return
a;
}
void addArc(int u, int v, int w){
  Arc* a = makeArc(v,w);
  a->next = A[u]; A[u] = a;
}
```

```
void input(){
  scanf("%d%d",&n,&m);
  for(int v = 1; v <= n; v++) A[v] = NULL;
  for(int k = 1; k <= m; k++){
    int u,v,w;
    scanf("%d%d%d",&u,&v,&w);
    addArc(u,v,w);
  }
  scanf("%d%d",&s,&t);
}
```

THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC PRIORITY QUEUE

```
void initPQ(){
  sH = 0;
  for(int v = 1; v <= n; v++)
    idx[v] = -1;
}
```

```
int main(){
   input();
   solve();
   return 0;
}
```

```
void solve(){
    for(int v = 1; v <= n; v++) d[v] = INF;
    initPQ(); pushPQ(s,0);
    while(!pqEmpty()){
        int u = deleteMin();
        for(Arc* a = A[u]; a != NULL; a = a->next){
            int v = a->id; int w = a->w;
            if(d[v] > d[u] + w)      pushPQ(v,d[u]+w);
        }
    }
    int rs = d[t]; if(d[t]==INF) rs = -1;
    printf("%d",rs);
}
```

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !