

# Grafuri (Graphs)

## Reprezentări

- Matrice de adiacență: /də:ə/
- Liste de adiacență: `vector<Node> list[NodeCount];`
- Lista muchiilor/arcilor: `vector<pair<Node, Node>> list;`

## Grafuri neorientate (Undirected Graphs)

Noduri (Vertices); Muchii (Edges)

Graf complet (Complete Graph): toate muchiile adiacente

Graf partial (partial?): ștergem muchii

Subgraf (Subgraph): ștergem noduri

Lanț (Walk / Chain): adiacente 2 cate 2

Lanț elementar (Trail): lanț cu noduri distincte

Ciclu (Cycle): lanț cu primu' = ultimu'

Ciclu elementar (elementary?): You get the point

Graf aciclic (Acyclic): natürlich

Conex (Connected):  $\forall v, w \exists$  lanț

Componenta conexă (Connected component): subgraf maximal (aka tăți bucata)

Lanț eulerian (Eulerian Path/ just 'Path'): lanț care vizitează fiecare muchie o singura dată

Ciclu eulerian (Eulerian cycle/circuit): możesz to rozgryźć

Graf eulerian (Eulerian Graph): Graf cu ciclu eulerian

Bipartit (Bipartite): putem împărți nodurile în 2 partiții, în care 2 noduri din aceeași partiție nu sunt adiacente

Lanț hamiltonian (Hamiltonian path): lanț elementar cu toate nodurile

Ciclu hamiltonian (Hamiltonian cycle): duh

## Grafuri orientate (Directed Graphs)

Vârfuri (Vertices); Arce (Edges/arc)

Grad interior: câte intră

Grad exterior: câte ies

Drum (Directed Walk): "lanț"

Graph Turneu (Tour): între oricare 2 varfuri distincte exista un arc

Slab Conex (Weakly connected): dacă-l transformam în neorientat, e conex

Tare Conex (Strongly connected): există drum între oricare 2 varfuri, dus-intors

Modalitati de parcurgere: Breadth first, Depth first

## Grafuri ponderate (Weighted Graphs)

### Dijkstra

```
1 function Dijkstra(Graph, source):
2     create vertex set Q
3     for each vertex v in Graph:
4         dist[v] = INFINITY
5         prev[v] = UNDEFINED
6         add v to Q
7     dist[source] = 0
```

```

8
9   while Q is not empty:
10       u = vertex in Q with min dist[u]
11       remove u from Q
12       for each neighbor v of u:           # only v that are still in Q
13           alt = dist[u] + length(u, v)
14           if alt < dist[v]:
15               dist[v] = alt
16               prev[v] = u
17
18   return dist[], prev[]

```

## Roy-Floyd

```

1   let dist be a  $|V| \times |V|$  array of minimum distances initialized to  $\infty$ 
2   let next be a  $|V| \times |V|$  array of vertex indices initialized to null
3
4   procedure FloydWarshallWithPathReconstruction():
5       for each edge (u, v) do
6           dist[u][v] = w(u, v) # The weight of the edge (u, v)
7           next[u][v] = v
8       for each vertex v do
9           dist[v][v] = 0
10          next[v][v] = v
11      for k from 1 to |V| do # standard Floyd-Warshall implementation
12          for i from 1 to |V|
13              for j from 1 to |V|
14                  if dist[i][j] > dist[i][k] + dist[k][j] then
15                      dist[i][j] = dist[i][k] + dist[k][j]
16                      next[i][j] = next[i][k]
17
18  procedure Path(u, v)
19      if next[u][v] = null then
20          return []
21      path = [u]
22      while u  $\neq$  v
23          u = next[u][v]
24          path.append(u)
25      return path

```

## Arbori (Trees)

Subarbori (Subtree): fii

Nivelul unui nod (Level): distanta pana la radacina

Inaltimea unui nod (Height): cel mai lung lant pana la un nod terminal

Arbore partial al unui graf: graf partial care este arbore

Arbore partial de cost minim (minimum-spanning-tree): avem o functie  $f$  care determina costul fiecarei muchii, gasim un arbore partial aî suma costurilor sale sa fie minima.

## Reprezentări

- like a graph: /də:ə/
- Ref descendente: `struct Node { int data; Vector<Node*> children; };`
- Ref ascendente: `int parinte[NodeCount] = {-1, ...};`

## Kruskal

```
1 algorithm Kruskal(G) is
2   A := ∅
3   for each v ∈ G.V do
4     MAKE-SET(v)
5   for each (u, v) in G.E ordered by weight(u, v), increasing do
6     if FIND-SET(u) ≠ FIND-SET(v) then
7       A := A ∪ {(u, v)}
8       UNION(FIND-SET(u), FIND-SET(v))
9   return A
```

## Prim

```
1 void prim(double mat[sz][sz], ssize_t len) {
2     int* s = new int[len] {-1}, j;
3     for (int i = 1; i < len; ++i) s[i] = 0;
4     for (int k = 1; k < len; ++k) {
5         double min = inf;
6         for (int i = 0; i < len; ++i)
7             if (s[i] != -1 && min > mat[i][s[i]]) {
8                 min = mat[i][s[i]];
9                 j = i;
10            }
11        std::cout << "edge: " << j << "-" << s[j] << "\n";
12        for (int i = 0; i < len; ++i)
13            if (s[i] != -1 && mat[i][s[i]] > mat[i][j])
14                s[i] = j;
15        s[j] = -1;
16    }
17 }
```

## Arbori binari (Binary Tree)

Arbori binari completi (Perfect binary tree): pe fiecare nivel  $s$  are exact  $2^s$  noduri

Aproape complet (Complete binary tree): pe ultimul nivel lipsesc doar primele din stanga/ ultimele din dreapta

Ansamblul Heap binar minim (Min-heap): Arbore aproape complet cu cheia oricarui parinte mai mica sau egala cu a fiului

Heap sort: stergem si punem intr-un vector

Coadă de priorități (Priority queue): tl;dr a fancy heap

## AVL Trees

```
1 struct Node {
2     Key key;
3     Node *left, *right;
```

```

4     int h = 1;
5     static int height(const Node* p) { return (p == nullptr) ? 0 : p->h; }
6     void updateHeight() { h = std::max(height(left), height(right)) + 1; }
7     int balanceFactor() const { return height(left) - height(right); }
8     static void rightRotate(Node*& n) {
9         Node* t = n->left;
10        n->left = t->right;
11        t->right = n;
12        n->updateHeight();
13        t->updateHeight();
14        n = t;
15    }
16    static void leftRotate(Node*& n) { /* like above but swap right and left*/ }
17    static void balance(Node*& node) {
18        node->updateHeight();
19        int factor = node->balanceFactor();
20        if (factor > 1) {
21            if (node->left->balanceFactor() < 0) leftRotate(node->left);
22            rightRotate(node);
23        } else if (factor < -1) {
24            if (node->right->balanceFactor() > 0) rightRotate(node->right);
25            leftRotate(node);
26        }
27    }
28    static void add(Node*& node, const Key& key) {
29        if (node == nullptr) { node = new Node(key); return; }
30        assert(key != node->key, "Value already in tree");
31        if (key < node->key) add(node->left, key);
32        else if (key > node->key) add(node->right, key);
33        balance(node);
34    }
35
36    static void remove(Node*& n, const Key& key) {
37        if (n == nullptr) { printf("not found?\n"); return; }
38        if (key < n->key) remove(n->left, key);
39        else if (key > n->key) remove(n->right, key);
40        else { // key == n->key
41            if (n->left == nullptr) {
42                if (n->right == nullptr) { n = nullptr; return; }
43                Node* tmp = n;
44                n = n->right;
45                tmp->right = nullptr;
46                delete tmp;
47            } else if (n->right == nullptr) {
48                Node* tmp = n;
49                n = n->left;
50                tmp->left = nullptr;
51                delete tmp;
52            } else {
53                constexpr auto findMax = [](Node*& r, auto& findMax) -> Node*& {
54                    if (r->right == nullptr) return r;
55                    return findMax(r->right, findMax);

```

```

56         };
57         Node*& tmp = findMax(n->left, findMax);
58         n->key = tmp->key;
59         remove(n->left, tmp->key);
60     }
61 }
62 if (n == nullptr) return;
63 balance(n);
64 }
65 };

```

## Binary heap

```

1  Vector<Val> data;
2  constexpr static int parent(int i) { return (i - 1) / 2; }
3  constexpr static int left(int i) { return (2 * i + 1); }
4  constexpr static int right(int i) { return (2 * i + 2); }
5  constexpr bool empty() const { return data.empty(); }
6  void push(const Val& v) {
7      data.push_back(v);
8      for (size_t i = data.size()-1; ;) {
9          if (i == 0) break;
10         size_t j = parent(i);
11         if (data[i] > data[j]) {
12             std::swap(data[i], data[j]);
13             i = j;
14         }
15     }
16 }
17 Val pop() {
18     Val res = std::move(data[0]);
19     data[0] = data.back();
20     data.pop_back();
21     size_t i = 0;
22     size_t sz = data.size();
23     while (i < sz) {
24         size_t j = left(i);
25         if (j < sz) {
26             if (j+1 < sz && data[j+1] > data[j]) ++j;
27             if (data[i] < data[j]) {
28                 std::swap(data[i], data[j]);
29                 i = j;
30             } else break;
31         } else break;
32     }
33     return res;
34 }

```

## Tablele de dispersie (Hash tables)

functii:  $h(k) = k \bmod m$ ;  $h(k) = \lfloor m(k\varphi - [k\varphi]) \rfloor$ ,  $\varphi = \frac{\sqrt{5}-1}{2}$

Or: do it again:  $h(k, i) = h_1(k) + i \bmod m$ ;  $h(k, i) = h_1(k) + c_1i + c_2i^2 \bmod m$

## Binary search trees

```
1 void insert(Node*& root, int key, int value) {
2     if (!root)
3         root = new Node(key, value);
4     else if (key == root->key)
5         root->value = value;
6     else if (key < root->key)
7         insert(root->left, key, value);
8     else // key > root->key
9         insert(root->right, key, value);
10 }
11 constexpr NodeRef findRef(const Key& key) {
12     auto impl = [&key] (Node*& t, auto& impl) {
13         if (t == nullptr || key == t->key) return NodeRef { t};
14         if (key < t->key) return impl(t->left, impl);
15         return impl(t->right, impl);
16     };
17     return impl(root, impl);
18 }
19
20 constexpr void remove(NodeRef r) {
21     assert(r.val != nullptr, "Node not found");
22     Node*& n = r.val;
23     if (n->left == nullptr) {
24         if (n->right == nullptr) { n = nullptr; return; }
25         Node* tmp = n;
26         n = n->right;
27         tmp->right = nullptr;
28         delete tmp;
29     } else if (n->right == nullptr) {
30         Node* tmp = n;
31         n = n->left;
32         tmp->left = nullptr;
33         delete tmp;
34     } else {
35         constexpr auto findMax = [] (Node*& r, auto& findMax) -> Node*& {
36             if (r->right == nullptr) return r;
37             return findMax(r->right, findMax);
38         };
39         Node*& replacement = findMax(n->left, findMax);
40         Node* tmp = replacement;
41         std::swap(replacement->key, n->key);
42         std::swap(replacement->val, n->val);
43         replacement = replacement->left;
44         tmp->right = tmp->left = nullptr;
45         delete tmp;
46     }
47 }
48 constexpr void remove(const Key& key) { remove(findRef(key)); }
```