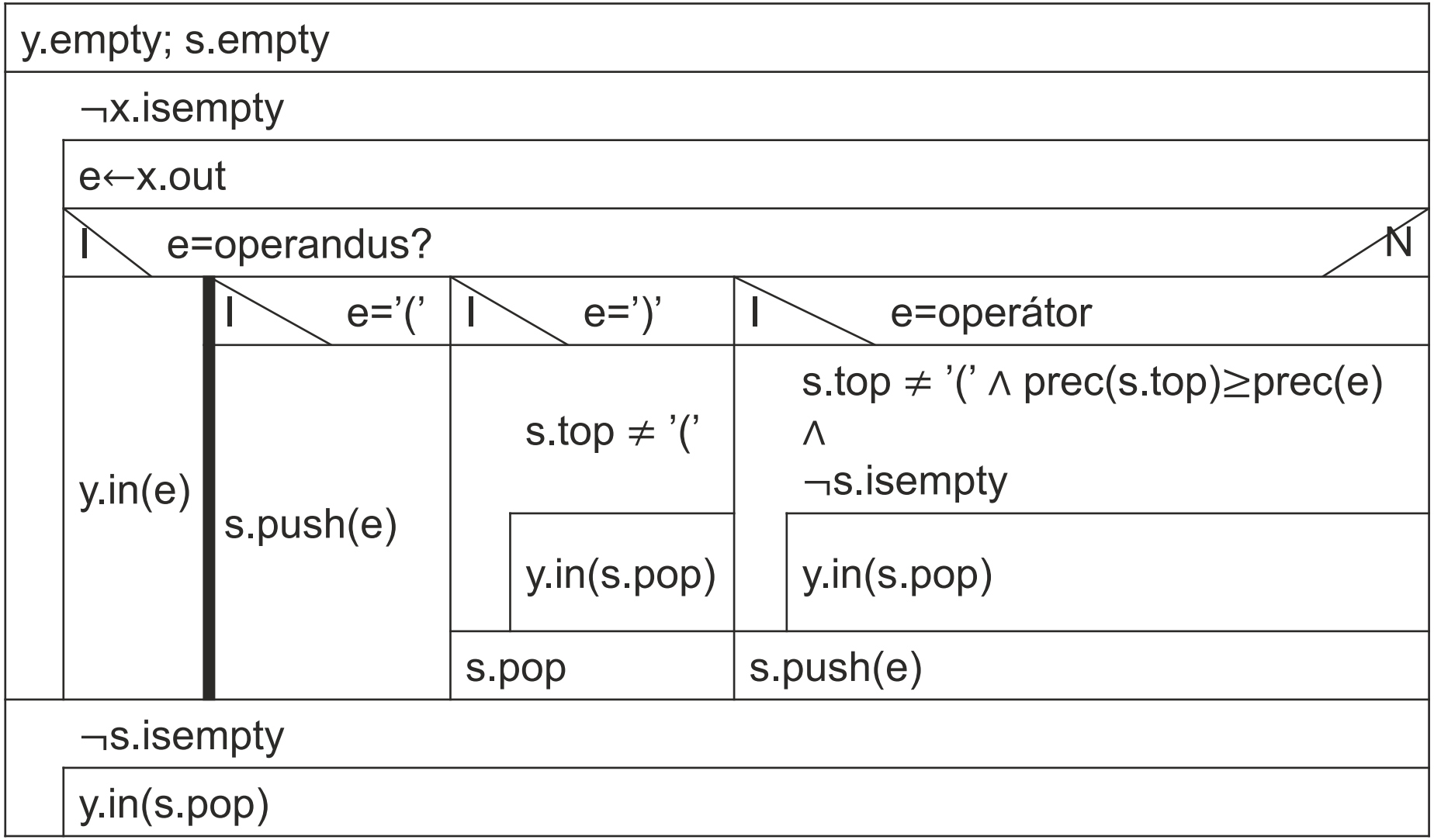
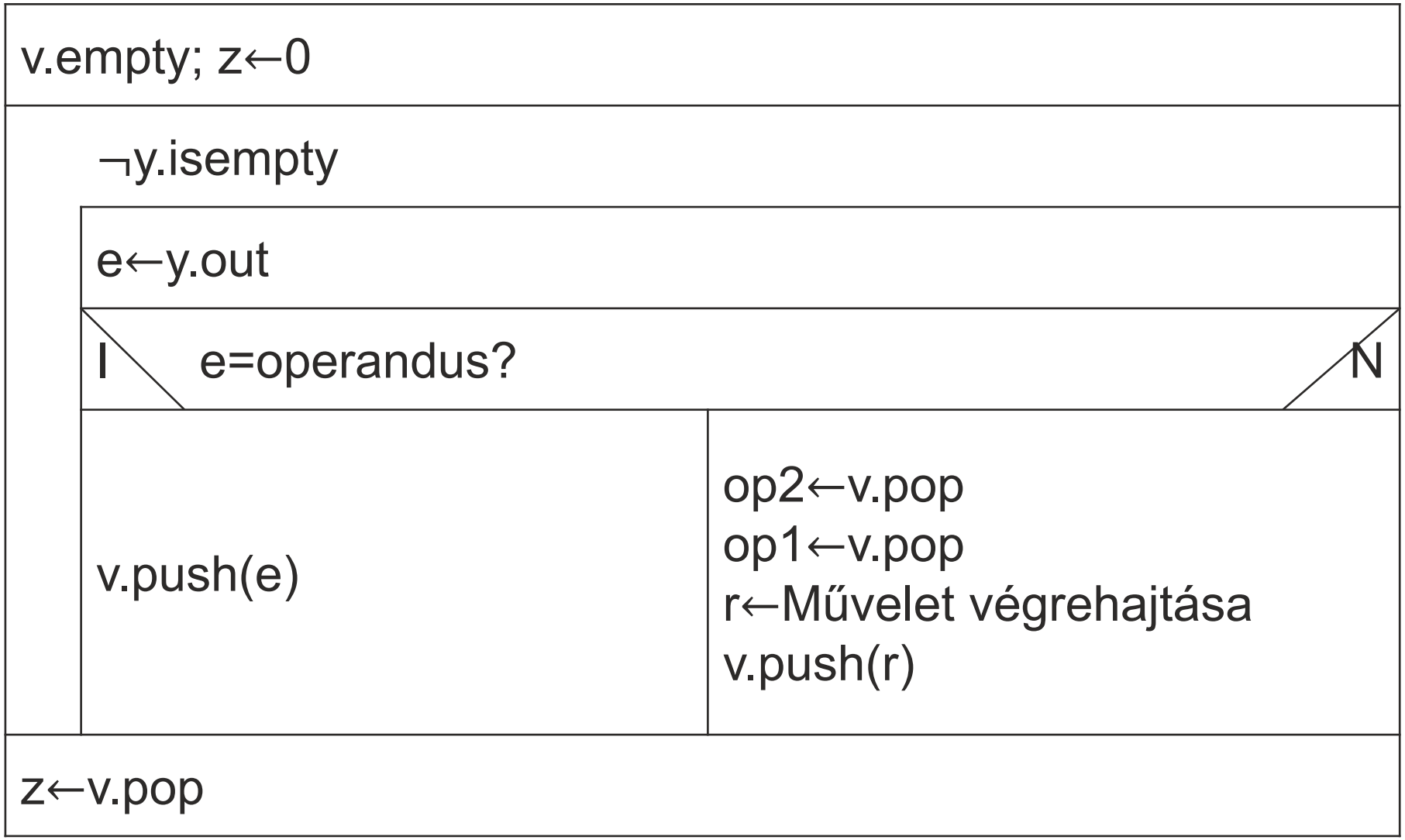
# ADATSZERKEZETEK ÉS ALGORITMUSOK

Lengyelforma, Kupac, Prioritásos Sor

# Lengyel formára alakítás

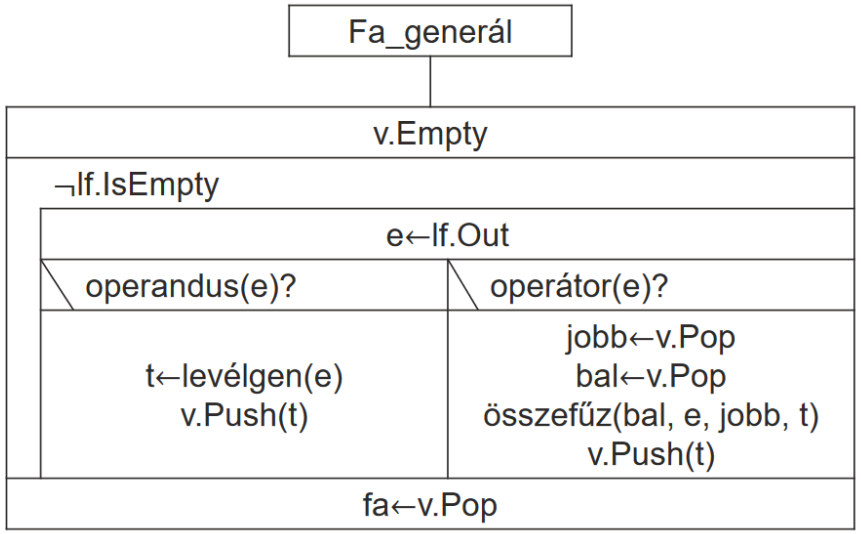


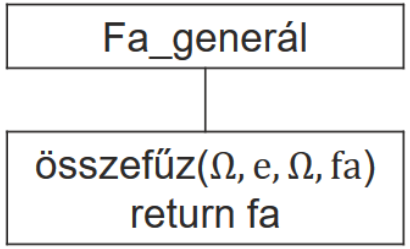
# Lengyel forma kiértékelése

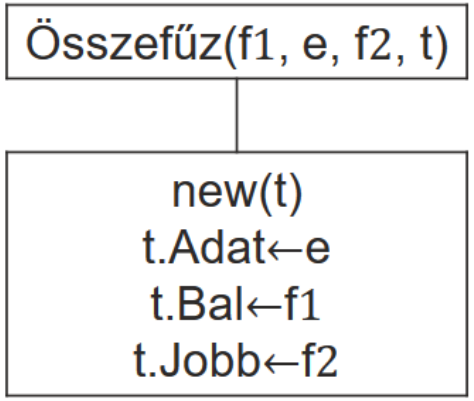


# Lengyelforma és kifejezés fák

- Használjuk fel a meglévő Lengyelforma, Queue és Stack implementációkat (letölthető Moodle-ből)
- Valósítsuk meg az ExpressionTree osztályt, mely egy fordított lengyel formát tartalmazó string-ből kifejezés fát épít majd inorder kiolvassa
- Használjuk az elméleten tanult algoritmust:



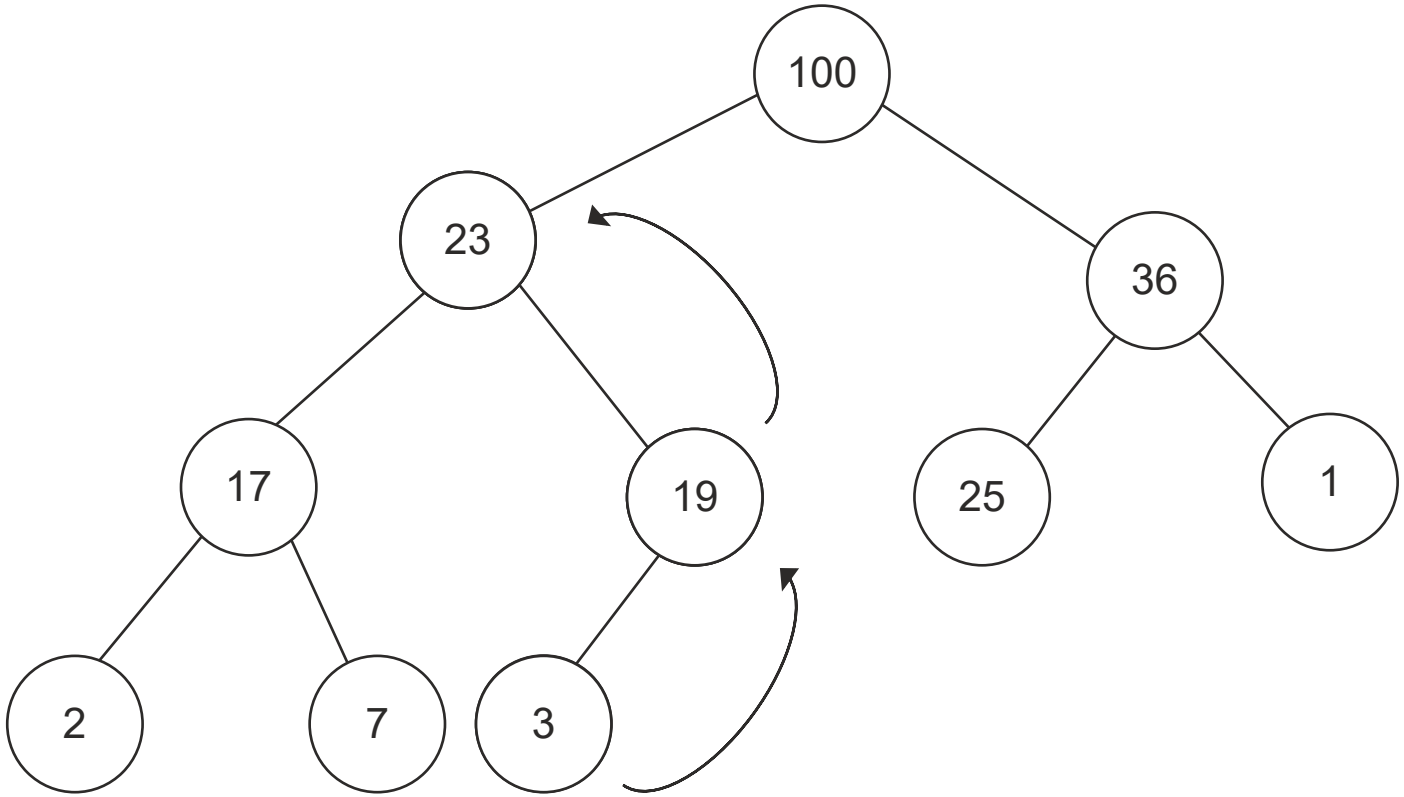



# Beszúrás közelebbről

- Miután az elemet beszúrtuk az első szabad helyre, biztosítanunk kell, hogy a fa továbbra is megfelel a kupactulajdonságnak, ezért a frissen beszúrt elemet a helyére juttatjuk, „felbuborékoltatjuk” a fában.
- Buborékoltatás: az elemet a fában felfelé / lefelé mozgatjuk.

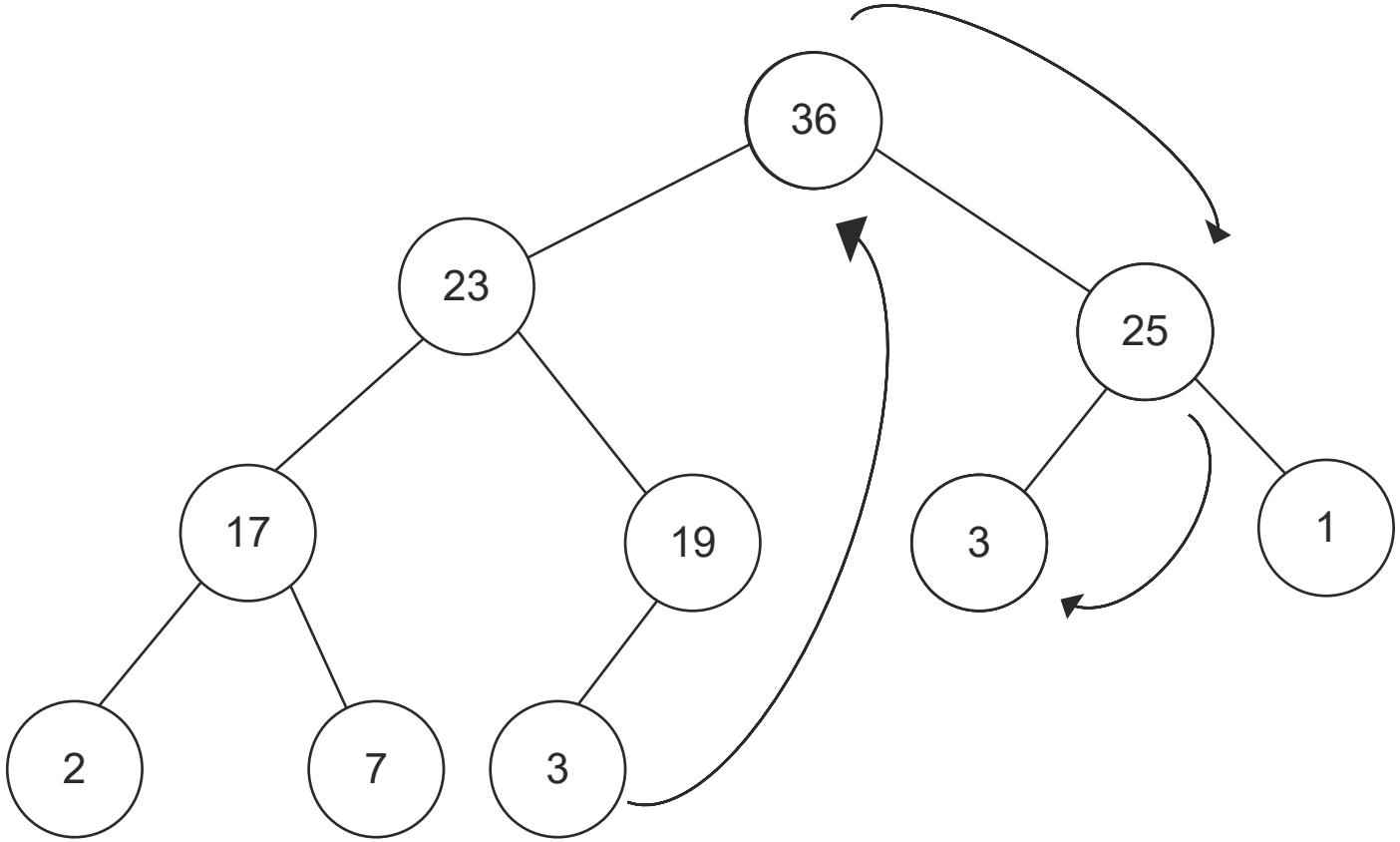
# Beszúrás



# Törlés közelebbről

- Elmentjük a gyökérelemet, majd az utolsó elemet kivesszük a helyéről, és a gyökércsúcsba tesszük.
- A kupactulajdonság teljesülését most is biztosítanunk kell, ezért a gyökérbe került elemet „lefelé buborékoltatjuk”, mindig a **nagyobb** gyerek felé, amíg a helyére nem kerül.
- Az elmentett gyökérelemből csak ezután lesz visszatérési érték.

# Törlés



# Prioritásos sor

Priority queue

# Prioritásos sor

- Egy olyan hagyományos sor, melyben minden elem ki van egészítve egy prioritást jelző adattaggal.
- A sor `out` művelete mindig a legnagyobb prioritású elemet adja vissza.
- Megvalósítható még kétirányú láncolt listával is.
  - Ilyenkor a beszúrásnál a prioritás szerinti helyére szúrjuk be az elemet
  - Ez lassabb az átlagos esetben

| Művelet | Kupac | Láncolt lista |
|---|---|---|
| Beszúrás | O(log(*n*)) | O(*n*) |
| Törlés | O(log(*n*)) | O(1) |

# Prioritásos sor megvalósítása kupaccal

- A prioritásos sort megvalósíthatjuk kupaccal is:
  - Ekkor a kupacban az elemek mellett tudnunk kell azok prioritását is.
    - Ezt tárolhatjuk egy belső `Node` típusban.
      - Mi ezt fogjuk csinálni.
    - Vagy megadható comparator segítségével.
      - A `std::priority_queue` ezt teszi.
  - A kupac két elemet a prioritása alapján hasonlít össze:
    - mindig a legnagyobb prioritású elem van felül (az kerül ki elsőnek a kupacból).
  - Ennek megfelelően kell megírni a `Node` összehasonlító operátorát.

# std::priority_queue

- STL container:
  - `std::vector`, `std::map`, stb
- Container adaptor: az `std::priority_queue` egy úgynevezett adapter, ami egy szokásos containert „csomagol be”, és konstans időben teszi elérhetővé a legmagasabb prioritású elemet
  - Alapértelmezésben `std::vector`-t használ

```
#include <queue>
int main() {
  std::priority_queue<int> q;
  for (int n : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
    q.push(n);

  std::priority_queue<int, std::vector<int>, std::greater<int>> q2;
  for (int n : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
    q2.push(n);
}
```

# Órai kódolás

- Használjuk fel a kupacot a prioritásos sor megvalósításához!