

# SCC-210

# Algoritmos Avançados

## Capítulo 2

## Estruturas de Dados

## Lineares - Parte 2

João Luís G. Rosa

# Fila de Prioridade

- ◆ É uma fila na qual os elementos de maior prioridade são removidos primeiro.
- ◆ Na implementação em STL, os elementos de maior prioridade são aqueles de maior valor.
- ◆ Filas de prioridade não possuem iteradores. Portanto, não se pode percorrer a fila e processar os elementos.

# Fila de Prioridade

## ◆ Características:

- Inserção e remoção em tempo logarítmico;
- Retorno do elemento de maior prioridade em tempo constante.

◆ Portanto, tem-se uma implementação por *heaps*.

# Fila de Prioridade

- ◆ São úteis, por exemplo, para:
  - Implementar algoritmo de Dijkstra;
  - Implementar Prim e Kruskal;
  - Implementar buscas heurísticas como  $A^*$ ;
  - Implementar métodos de ordenação.

# Fila de Prioridade

## ◆ Filas de prioridade

- `#include <queue>`
- Declaração: `priority_queue <T>`
- Operações:
  - ◆ `q.size()` – Fornece o número de elementos na fila de prioridade;
  - ◆ `q.empty()` – Fornece true se a fila de prioridade estiver vazia;
  - ◆ `q.push(E)` – Insere **E** no final da fila;
  - ◆ `q.top()` – Retorna o elemento no topo da fila de prioridade (i.e. o elemento de maior valor);
  - ◆ `q.pop()` – Remove o elemento no topo da fila de prioridade;
  - ◆ `q1 == q2` – true se **q1** e **q2** possuem os mesmos elementos.

# Exemplo: Fila de Prioridade

```
#include<cstdio>
#include<queue>

using namespace std;

main() {
    priority_queue<int> pq;

    pq.push(5);
    pq.push(1);
    pq.push(3);

    printf("%d\n", pq.top());
    pq.pop();
    printf("%d\n", pq.top());
    pq.pop();
    printf("%d\n", pq.top());
    pq.pop();

    if (pq.empty())
        printf("Fila vazia\n");
}
```

# Exemplo: Fila de Prioridade

```
#include<cstdio>
#include<queue>

using namespace std;

main() {
    priority_queue<int, vector<int>, greater<int> > pq;

    pq.push(5);
    pq.push(1);
    pq.push(3);

    printf("%d\n", pq.top());
    pq.pop();
    printf("%d\n", pq.top());
    pq.pop();
    printf("%d\n", pq.top());
    pq.pop();

    if (pq.empty())
        printf("Fila vazia\n");
}
```

# Exemplo: Fila de Prioridade

```
#include<cstdio>
#include<queue>

using namespace std;

struct aresta {
    int fonte, destino, peso;

    bool operator<(const aresta &v) const {
        return peso < v.peso;
    }
};

int main() {
    priority_queue<aresta> pq;
    aresta v, u;

    v.fonte = 1; v.destino = 2; v.peso = 10;
    pq.push(v);
    v.fonte = 3; v.destino = 5; v.peso = 5;
    pq.push(v);

    u = pq.top();
    printf("%d\n", u.peso);
}
```



# Conjuntos

- ◆ São *containers* ordenados que permitem o uso simples de algoritmos de união, intersecção, e cálculo de diferenças.
- ◆ Por serem conjuntos ordenados, é importante definir uma relação de ordem ( $<$ ).
- ◆ Existem duas classes containers:
  - **set**: conjunto cujas chaves não podem se repetir;
  - **multiset**: “conjunto” cujas chaves podem se repetir.

# sets

- **#include <set>**
- Declaração: **set <T>**
- Operações:
  - ♦ **s.insert(E)** - Insere **E** no conjunto, se **E** já pertence ao conjunto, nada é feito;
  - ♦ **s.erase(E)** - Remove **E** do conjunto, se **E** não já pertence ao conjunto, nada é feito;
  - ♦ **s.find(E)** - Localiza **E** no conjunto. Se **E** não pertence ao conjunto, retorna **s.end()**;
  - ♦ **s.size()** - Fornece o número de elementos do conjunto;
  - ♦ **s.empty()** - Retorna true se o conjunto for vazio;
  - ♦ **s.clear()** - Apaga todos os elementos do conjunto;
  - ♦ **s1 == s2** - true se **s1** e **s2** possuírem os mesmos elementos;

# sets

## ■ Continuação:

- ◆ **s.count(E)** – Retorna o número de elementos cuja chave é igual a **E** (resposta 0 ou 1);
- ◆ **s.lower\_bound(E)** – Retorna o primeiro elemento cuja chave é não menor do que **E**;
- ◆ **s.upper\_bound(e)** – Retorna o primeiro elemento cuja chave é maior do que **E**.

# sets - Exemplo

```
#include<set>
#include<cstdio>;

using namespace std;

int main() {
    set<int> a;
    set<int>::iterator i;

    a.insert(1); a.insert(3);
    a.insert(5); a.insert(3);

    for (i=a.begin(); i!=a.end(); i++)
        printf("%d ", *i);           // 1 3 5 ordenado
}
```

# multisets

- `#include <set>`
- Declaração: `multiset <T>`
- Operações:
  - ♦ `s.insert(E)` - Insere **E** no conjunto, se **E** já pertence ao conjunto, passa a existir uma outra cópia de **E**;
  - ♦ `s.erase(E)` - Remove **E** do conjunto, se **E** não já pertence ao conjunto, nada é feito;
  - ♦ `s.find(E)` - Localiza **E** no conjunto. Se **E** não pertence ao conjunto, retorna `s.end()`;
  - ♦ `s.size()` - Fornece o número de elementos do conjunto;
  - ♦ `s.empty()` - Retorna true se o conjunto for vazio;
  - ♦ `s.clear()` - Apaga todos os elementos do conjunto;
  - ♦ `s1 == s2` - true se **s1** e **s2** possuírem os mesmos elementos;

# multisets

- Continuação:

- ◆ `s.count(E)` – Retorna o número de elementos cuja chave é igual a `E`;
- ◆ `s.lower_bound(E)` – Retorna o primeiro elemento cuja chave é não menor do que `E`;
- ◆ `s.upper_bound(E)` – Retorna o primeiro elemento cuja chave é maior do que `E`.

# multisets - Exemplo

```
#include<set>
#include<cstdio>;

using namespace std;

int main() {
    multiset<int> a;
    multiset<int>::iterator i;

    a.insert(1); a.insert(3);
    a.insert(5); a.insert(3);

    for (i=a.begin(); i!=a.end(); i++)
        printf("%d ", *i);           // 1 3 3 5 ordenado
}
```

# Algoritmos sobre conjuntos

- ◆ Tanto para **set** quanto para **multiset**, STL provê alguns algoritmos clássicos de manipulação.
- ◆ Esses algoritmos não são métodos dessas classes, mas funções declaradas no arquivo de inclusão: *algorithm* (**`#include<algorithm>`**).



# Algoritmos sobre conjuntos

◆ As principais funções são:

- **set\_union** - Realiza a união entre dois conjuntos:

- ◆ **set**: união clássica entre conjuntos;
- ◆ **multiset**: no caso de elementos repetidos  $\max(m,n)$ ;
- ◆ Complexidade linear.

- **set\_intersection** - Realiza a interseção entre conjuntos:

- ◆ **set**: intersecção clássica entre conjuntos;
- ◆ **multiset**: no caso de elementos repetidos  $\min(m,n)$ ;
- ◆ Complexidade linear.

# Algoritmos sobre conjuntos

## ◆ Continuação:

- **set\_difference** – Realiza a diferença entre dois conjuntos:
  - ◆ **set**: diferença clássica entre conjuntos:
    - $A - B =$  elementos em A que não ocorrem em B.
  - ◆ **multiset**: no caso de elementos repetidos  $\max(m-n, 0)$ ;
  - ◆ Complexidade linear.

# multisets - Exemplo

```
#include<algorithm>
#include<set>
#include<cstdio>;

using namespace std;

int main() {
    set<int> a, b, c, d, e;

    a.insert(1); a.insert(3); a.insert(5);
    b.insert(2); b.insert(3); b.insert(6);

    set_union(a.begin(), a.end(), b.begin(), b.end(),
              inserter(c, c.begin()));

    set_intersection(a.begin(), a.end(), b.begin(), b.end(),
                    inserter(d, d.begin()));

    set_difference(a.begin(), a.end(), b.begin(), b.end(),
                  inserter(e, e.begin()));
}
```

# Conjuntos como Arranjo de Bits

- ◆ Arranjo binário de  $n$  posições  $\rightarrow$  subconjuntos de  $n$  elementos:
  - Bit  $i = 1$   $\rightarrow$   $i$ -ésimo elemento pertence ao subconjunto.
  - Bit  $i = 0$   $\rightarrow$  caso contrário.
- ◆ Representação mais simples e conveniente para subconjuntos derivados de universos estáticos e de tamanho moderado.
- ◆ Inserção e remoção por chaveamento de bit.
- ◆ Interseção e união via operações lógicas AND e OR nos arranjos.
- ◆ Eficiente em termos de memória:
  - Arranjo de 1000 inteiros de 4 bytes pode representar qualquer subconjunto de 32000 elementos!

# Exercício: Hartals

PC/Uva IDs: 110203/10050, Popularity: B, Sucess rate: high, Level: 2

Political parties in Bangladesh show their muscle by calling for regular *hartals* (strikes), which cause considerable economic damage. For our purposes, each party may be characterized by a positive integer  $h$  called the *hartal parameter* that denotes the average number of days between two successive strikes called by the given party.

Consider three political parties. Assume  $h_1 = 3$ ,  $h_2 = 4$ , and  $h_3 = 8$ , where  $h_i$  is the hartal parameter for party  $i$ . We can simulate the behavior of these three parties for  $N = 14$  days. We always start the simulation on a Sunday. There are no hartals on either Fridays or Saturdays.

# Exercício: Hartals

Days	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
Party 1			x			x			x			x		
Party 2				x				x				x		
Party 3								x						
Hartals			1	2				3	4			5		

There will be exactly five hartals (on days 3, 4, 8, 9, and 12) over the 14 days. There is no hartal on day 6 since it falls on Friday. Hence we lose five working days in two weeks.

Given the hartal parameters for several political parties and the value of  $N$ , determine the number of working days lost in those  $N$  days.

# Exercício: Hartals

## ◆ Input

The first line of the input consists of a single integer  $T$  giving the number of test cases to follow. The first line of each test case contains an integer  $N$  ( $7 \leq N \leq 3,650$ ), giving the number of days over which the simulation must be run. The next line contains another integer  $P$  ( $1 \leq P \leq 100$ ) representing the number of political parties. The  $i$ th of the next  $P$  lines contains a positive integer  $h_i$  (which will never be a multiple of 7) giving the *hartal parameter* for party  $i$  ( $1 \leq i \leq P$ ).

## ◆ Output

For each test case, output the number of working days lost on a separate line.

# Referências

- ◆ Batista, G. & Campello, R.
  - Slides disciplina *Algoritmos Avançados*, ICMC-USP, 2007.
- ◆ Skiena, S. S. & Revilla, M. A.
  - *Programming Challenges - The Programming Contest Training Manual*. Springer, 2003.