SCC-211
Lab. 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

```cpp
#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

```cpp
#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 *hi* 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 Su	2 Mo	3 Tu	4 We	5 Th	6 Fr	7 Sa	8 Su	9 Mo	10 Tu	11 We	12 Th	13 Fr	14 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 ith 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.

118-Mutant_Flatworld_Explorers

- **Background**
  - Robotics, robot motion planning, and machine learning are areas that cross the boundaries of many of the subdisciplines that comprise Computer Science: artificial intelligence, algorithms and complexity, electrical and mechanical engineering to name a few. In addition, robots as ``turtles'' (inspired by work by Papert, Abelson, and diSessa) and as ``beeper-pickers'' (inspired by work by Pattis) have been studied and used by students as an introduction to programming for many years.
  - This problem involves determining the position of a robot exploring a pre-Columbian flat world.

118-Mutant_Flatworld_Explorers

The Problem

- Given the dimensions of a rectangular grid and a sequence of robot positions and instructions, you are to write a program that determines for each sequence of robot positions and instructions the final position of the robot.
- A robot *position* consists of a grid coordinate (a pair of integers: x-coordinate followed by y-coordinate) and an orientation (N,S,E,W for north, south, east, and west). A robot *instruction* is a string of the letters '*L*', '*R*', and '*F*' which represent, respectively, the instructions:
  - *Left*: the robot turns left 90 degrees and remains on the current grid point.
  - *Right*: the robot turns right 90 degrees and remains on the current grid point.
  - *Forward*: the robot moves forward one grid point in the direction of the current orientation and mantains the same orientation.

118-Mutant_Flatworld_Explorers

- The direction *North* corresponds to the direction from grid point (*x*,*y*) to grid point (*x*,*y*+1).
- Since the grid is rectangular and bounded, a robot that moves ``off'' an edge of the grid is lost forever. However, lost robots leave a robot ``scent'' that prohibits future robots from dropping off the world at the same grid point. The scent is left at the last grid position the robot occupied before disappearing over the edge. An instruction to move ``off'' the world from a grid point from which a robot has been previously lost is simply ignored by the current robot.

118-Mutant_Flatworld_Explorers

The Input

- The first line of input is the upper-right coordinates of the rectangular world, the lower-left coordinates are assumed to be 0,0.
- The remaining input consists of a sequence of robot positions and instructions (two lines per robot). A position consists of two integers specifying the initial coordinates of the robot and an orientation (N,S,E,W), all separated by white space on one line. A robot instruction is a string of the letters 'L', 'R', and 'F' on one line.
- Each robot is processed sequentially, i.e., finishes executing the robot instructions before the next robot begins execution.
- Input is terminated by end-of-file.

118-Mutant_Flatworld_Explorers

- You may assume that all initial robot positions are within the bounds of the specified grid. The maximum value for any coordinate is 50. All instruction strings will be less than 100 characters in length.

The Output

- For each robot position/instruction in the input, the output should indicate the final grid position and orientation of the robot. If a robot falls off the edge of the grid the word ``LOST'' should be printed after the position and orientation.

118-Mutant_Flatworld_Explorers

- **Sample Input**
  - 5 3
  - 1 1 E
  - RFRFRFRF
  - 3 2 N
  - FRRFLLFFRRFLL
  - 0 3 W
  - LLFFFLFLFL
- **Sample Output**
  - 1 1 E
  - 3 3 N LOST
  - 2 3 S

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.