Heaps **binárias**

STL

Prof. Edson Alves – UnB/FGA

Sumário

Filas com Prioridades

Definição de filas com prioridades

- As filas com prioridades são variações da fila onde os elementos são acessados ou inseridos de acordo com a prioridade estabelecida
- Assim, a estratégia FIFO não se mantém: o primeiro elemento a sair não é mais o primeiro a entrar, e sim o elemento com maior prioridade presente na fila
- O desafio é encontrar uma implementação eficiente
- Se os elementos são inseridos ordenadamente na fila, de acordo com a prioridade, a complexidade do método `push()` é $O(N)$, e do método `pop()` é $O(1)$
- Se os elementos são inseridos no final da fila, e a prioridade é avaliada no momento do acesso, a complexidade do método `push()` é $O(1)$, e do método `pop()` é $O(N)$

Filas com prioridades em C++

- A STL do C++ oferece um contêiner que implementa uma fila com prioridades: a `priority_queue`, que faz parte da biblioteca queue
- Esta fila com prioridades é implementada através de uma *heap* binária
- Esta estratégia permite que os métodos `push()` e `pop()` sejam implementados com complexidade $O(\log N)$
- Diferentente da fila, para acessar o próximo elemento, segundo a prioridade estabelecida, é utilizado o método `top()`
- Por padrão o maior elemento, segundo o critério de comparação, é o de maior prioridade
- Este comportamento pode ser modificado através da reescrita do operador de comparação

Exemplo de filas com prioridades

```cpp
#include <bits/stdc++.h>

using namespace std;

struct Paciente {
    string nome; int idade;

    Paciente(const string& n, int a) : nome(n), idade(a) {}

    bool operator<(const Paciente& p) const {
        // Idosos tem maior prioridade, mais velhos primeiro
        if (idade >= 65) return idade < p.idade;

        // Depois crianças, mais novas primeiro
        if (idade <= 6) return p.idade >= 65 or (p.idade <= 6 and idade > p.idade);

        // Os demais por idade
        return p.idade >= 65 or p.idade <= 6 or p.idade > idade;
    }
};
```

Exemplo de filas com prioridades

```cpp
int main()
{
    priority_queue<Paciente> pq;

    pq.push(Paciente("Maria", 25));
    pq.push(Paciente("Carlos", 70));
    pq.push(Paciente("Neto", 4));
    pq.push(Paciente("Laura", 25));
    pq.push(Paciente("Beatriz", 32));
    pq.push(Paciente("Pedro", 5));
    pq.push(Paciente("Beto", 68));

    while (not pq.empty()) {
        auto p = pq.top(); pq.pop();
        cout << p.nome << ": " << p.idade << " anos\n";
    }

    return 0;
}
```

Min heap **na STL**

- A `priority_queue` da STL é uma *max heap*
- Ela pode ser transformada em uma *min heap* de duas maneiras
- A primeira dela é útil quando a fila armazena tipos numéricos
- Neste caso, basta inserir o simétrico de cada elemento na fila, o que inverterá o critério de comparação
- Deve-se, contudo, tomar o cuidado de utilizar o simétrico do elemento máximo, quando este for extraído
- A segunda maneira é utilizar a estrutura `greater()` da STL, a qual faz com que o operator > seja utilizado nas comparações de ordenação
- Se a classe ou estrutura não tiver tal operador por padrão, o mesmo deve ser implementado

Exemplo de min heap na STL

```cpp
#include <bits/stdc++.h>

using namespace std;

int main()
{
    vector<int> xs { 40, 68, 15, 99, 24, 6, 51, 77 };

    priority_queue<int> p;

    for (auto x : xs)
        p.push(-x);

    while (not p.empty())
    {
        cout << -p.top() << ' ';
        p.pop();
    }

    cout << '\n';       // 6 15 24 40 51 68 77 99
```

Exemplo de min heap na STL

```
22      priority_queue<int, vector<int>, greater<int>> q(xs.begin(), xs.end());
23
24      while (not q.empty())
25      {
26          cout << q.top() << ' ';
27          q.pop();
28      }
29
30      cout << '\n';        // 6 15 24 40 51 68 77 99
31
32
33      return 0;
34  }
```

Ordenação e *heaps* na STL

Heaps e **ordenação**

- A biblioteca `algorithm` da linguagem C++ contém três rotinas de ordenação, a saber: `sort()`, `stable_sort()` e `partial_sort()`
- As *heaps* estão presentes em dois deles
- A função `sort()` é implementada através de uma estratégia mista: ela começa com o *Introsort* (que combina o *Quicksort* com o *Heapsort*) e finaliza com o *Insert sort*
- Já o `partial_sort()` é implementada por meio do *Heapsort*: é mantida uma *max heap* com exatamante k elementos, removendo o $(k+1)$-ésimo elemento sempre que o tamanho da *heap* for maior do que k
- Em seguida, os elementos são ordenados, retirando os elementos máximo, um por vez, e colocando-os nas posições apropriadas

Exemplo de uso das funções de ordenação em C++

```cpp
#include <bits/stdc++.h>

int main()
{
    std::vector<int> xs { 99, 40, 68, 15, 24, 77, 6, 51 };

    std::partial_sort(xs.begin(), xs.begin() + 3, xs.end());

    // xs = {6, 15, 24, 99, 68, 77, 40, 51}
    for (size_t i = 0; i < xs.size(); ++i)
        std::cout << xs[i] << (i + 1 == xs.size() ? "\n" : " ");

    std::sort(xs.begin(), xs.end(), std::greater<int>());

    // xs = {99, 77, 68, 51, 40, 24, 15, 6}
    for (size_t i = 0; i < xs.size(); ++i)
        std::cout << xs[i] << (i + 1 == xs.size() ? "\n" : " ");

    return 0;
}
```

Referências

1. CppReference. Priority Queue, acesso em 22/04/2019.
2. Geeks for Geeks. C `qsort()` vs C++ `sort()`, acesso em 22/04/2019.
3. Quora. What algorithm do popular C++ compilers use for `std::sort()` and `std::stable_sort()`, acesso em 22/04/2019.
4. Wikipedia. Introsort, acesso em 22/04/2019.