Estruturas de Dados usadas no projeto NeoLook

Node.h

Esta estrutura representa um nó em uma lista encadeada simples. Cada nó contém um valor e um ponteiro para o próximo nó na lista. Esta estrutura é usada na implementação da classe *Queue*.

Atributos

1. `Type data;` - Dado armazenado no Node.
2. `Node* next;` - Ponteiro para o próximo Node.

Métodos

Construtor

`Node(Type data, Node * next);` - Construtor da classe Node. Tem como parâmetros um valor e um ponteiro para o próximo node, que pode ser *nullptr*.

Destrutor

`~Node();` - Destrutor da classe Node.

Exemplo de implementação

```
Node<Type> *no = new Node<Type>(val, nullptr);
```

Queue.h

Uma fila é uma estrutura de dados linear que segue a ordem "Primeiro a Entrar, Primeiro a Sair" (FIFO - First In, First Out). Ela mantém uma coleção de elementos onde a remoção de elementos ocorre no início da fila e a inserção de elementos ocorre no final da fila. No âmbito do projeto Neolook, a estrutura de dados TAD

Queue será empregada na implementação do algoritmo FCFS, que adere ao princípio FIFO (First-In, First-Out).

Atributos

1. `Node<Type> *m_head;` - Ponteiro para o no sentinela.
2. `Node<Type> *m_tail;` - Ponteiro para o ultimo nó da fila.
3. `unsigned m_size{};` - Quantidade de elementos na fila.

Métodos

Construtor

`Queue();`: Inicializa uma fila vazia com o nó sentinela apontando para nulo, e o tamanho igual a zero.

```
#include <iostream>
#include "TAD'S/Queue.h"

int main() {
    Queue<int> q;  // declaração da Queue vazia
    std::cout << q.size() << std::endl;
}

// Saída: 0
```

Construtor de Cópia

`Queue(const Queue &lst);`: Recebe uma fila como parâmetro e cria uma nova fila copiando todos os elementos da fila original.

```
#include <iostream>

#include "TAD'S/Queue.h"

int main() {
    Queue<int> q1;  // declaração da Queue
    q1.push(1);
    q1.push(2);
    Queue<int> q2(q1);  // declaração de outra Queue, passando q1
                        // como parâmetro para cópia
    while (!q2.empty()) {
        std::cout << q2.front() << " ";
        q2.pop();
    }
```

```
}

// Saída: 1 2
```

Destrutor

`~Queue();`: Libera a memória alocada para a fila, incluindo o nó sentinela.

empty()

`bool empty() const;`: Verifica se a fila está vazia. Retorna `true` se o tamanho da fila for igual a zero (vazia), ou `false` caso contrário.

```
#include <iostream>

#include "TAD'S/Queue.h"

void queueVazia(Queue<int>& q) {
    if (q.empty()) std::cout << "Queue vazia" << std::endl;
    else std::cout << "Queue nao vazia" << std::endl;
}

int main() {
    Queue<int> q;
    queueVazia(q);
    q.push(1);
    queueVazia(q);
}

// Saída: Queue vazia
//        Queue nao vazia
```

size()

`unsigned int size() const;`: Retorna o número atual de elementos na fila, ou seja, o tamanho atual da fila.

```
#include <iostream>

#include "TAD'S/Queue.h"

int main() {
    Queue<int> q;
    std::cout << q.size() << " ";
    q.push(1);
    std::cout << q.size() << " ";
    q.push(2);
    q.push(3);
    std::cout << q.size() << " ";
```

```
    q.pop();
    std::cout << q.size() << " ";
}

// Saída: 0 1 3 2
```

clear()

`void clear();`: Remove todos os elementos da fila, deixando-a vazia.

```
#include <iostream>

#include "TAD'S/Queue.h"

int main() {
    Queue<int> q;
    std::cout << q.size() << " ";
    q.push(1);
    std::cout << q.size() << " ";
    q.push(2);
    q.push(3);
    std::cout << q.size() << " ";
    q.pop();
    std::cout << q.size() << " ";
    q.clear();
    std::cout << q.size() << " ";
}

// Saída: 0 1 3 2 0
```

push()

`void push(const int &val);`: Insere um elemento no final da fila. Recebe um valor como parâmetro e cria um novo nó com esse valor, inserindo-o na posição correta.

```
#include <iostream>

#include "TAD'S/Queue.h"

int main() {
    Queue<int> q;
    q.push(1);
    q.push(2);
    q.push(1);
    q.push(1234);

    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }
```

```
}

// Saída: 1 2 1 1234
```

front()

`int &front();`: Retorna uma referência para o primeiro elemento da fila. Lança uma exceção se a fila estiver vazia. Pode ser const.

```
#include <iostream>

#include "TAD'S/Queue.h"

int main() {
    Queue<int> q;
    q.push(35);
    q.push(2);
    q.push(1);
    q.push(1234);

    std::cout << q.front() << " ";

    q.pop();

    std::cout << q.front() << " ";
}

// Saída: 35 2
```

back()

`int &back();`: Retorna uma referência para o último elemento da fila. Lança uma exceção se a fila estiver vazia. Pode ser const.

```
#include <iostream>

#include "TAD'S/Queue.h"

int main() {
    Queue<int> q;
    q.push(35);
    q.push(2);
    q.push(1);
    q.push(1234);

    std::cout << q.back() << " ";
}
```

```
// Saída: 1234
```

Iteradores

begin()

`iterator_queue<Type> begin();` - Função que retorna um iterador que aponta para o primeiro elemento da fila.

end()

`iterator_queue<Type> end();` - Função que retorna um iterador que aponta para uma posição após o último elemento da fila.

```
#include <iostream>

#include "TAD'S/Queue.h"

int main() {
    Queue<int> q;
    q.push(35);
    q.push(2);
    q.push(1);
    q.push(1234);

    for (auto it = q.begin(); it != q.end(); ++it) {
        std::cout << *it << " ";
    }
}

// Saída: 35 2 1 1234
```

Exemplo de implementação

```
#include <iostream>

#include "TAD'S/Queue.h"

int main() {
    Queue<int> q;
    q.push(1);
    q.push(2);
```

```
    q.push(35);
    q.push(1234);
    std::cout << q.front() << " " << q.back() << " " << q.size() << " "
              << q.empty() << std::endl;
    return 0;
}

// Saída: 1 1234 4 0
```

PriorityQueue.h

Esta classe representa uma fila de prioridade baseada em uma min heap, sendo uma estrutura de dados que mantém uma coleção de elementos onde o elemento de menor valor (ou maior prioridade) está sempre no topo da fila. Ela é útil em situações em que se deseja garantir que o elemento de menor valor possa ser rapidamente recuperado e removido. A min heap é uma árvore binária completa em que cada nó pai tem um valor menor ou igual ao de seus nós filhos, garantindo que o elemento de menor valor esteja sempre na raiz.

A escolha da estrutura de dados TAD PriorityQueue no contexto do algoritmo SFJ (Shortest Job First) é justificada pela necessidade de ordenar a lista de processos com base no tempo de execução restante de cada processo, de forma a priorizar a execução dos mais curtos.

Atributos

1. `int* m_heap;` - Ponteiro para um vetor que armazena os elementos da fila de prioridade. Ele mantém os elementos em uma estrutura de *heap*, onde o elemento de maior prioridade é sempre o primeiro no vetor.
2. `unsigned m_size;` - Representa o tamanho atual da fila de prioridade, ou seja, a quantidade de elementos armazenados na fila.
3. `unsigned m_capacity;` - O atributo `m_capacity` especifica a capacidade máxima da fila de prioridade, ou seja, o número máximo de elementos que a fila pode conter sem a necessidade de realocação de memória.

Métodos

Construtor com parâmetro

`PriorityQueue(unsigned capacity)`: Construtor da classe. Cria uma instância da fila de prioridade com uma capacidade máxima especificada em `capacity`. Inicializa os atributos `m_capacity` e `m_size`, alocando memória para o vetor `m_heap`.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq(12);  // Cria uma fila de prioridade com capacidade 12

    std::cout << pq.size() << " ";

    pq.push(1);     // Insere o elemento 1 na fila
    pq.push(1234);  // Insere o elemento 1234 na fila
    pq.push(4);     // Insere o elemento 4 na fila
    pq.push(0);     // Insere o elemento 0 na fila

    std::cout << pq.size() << " ";
}

// Saída: 0 4
```

Construtor default

`PriorityQueue()`: Construtor padrão da classe. Cria uma instância da fila de prioridade com capacidade padrão de 1 elemento. Inicializa os atributos `m_capacity` e `m_size`, alocando memória para o vetor `m_heap`.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq;  // Cria uma Priotity Queue vazia, com capacidade
                            // para 1 elemento, do tipo inteiro

    std::cout << pq.size() << " ";

    pq.push(1);     // Insere o elemento 1 na fila
    pq.push(1234);  // Insere o elemento 1234 na fila
    pq.push(4);     // Insere o elemento 4 na fila
    pq.push(0);     // Insere o elemento 0 na fila

    std::cout << pq.size() << " ";
}

// Saída: 0 4
```

Destrutor

`~PriorityQueue()`: Destrutor da classe. Libera a memória alocada para o vetor `m_heap` quando a instância da fila de prioridade é destruída.

push()

`void push(Type key)`: Este método permite inserir um elemento na fila de prioridade. O elemento é inserido de acordo com sua prioridade na fila, e a estrutura da *heap* é mantida.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq(12);  // Cria uma fila de prioridade com capacidade 12

    pq.push(1);     // Insere o elemento 1 na fila
    pq.push(1234);  // Insere o elemento 1234 na fila
    pq.push(4);     // Insere o elemento 4 na fila
    pq.push(0);     // Insere o elemento 0 na fila

    while (!pq.empty()) {
        std::cout << pq.top() << " ";  // Imprime o elemento de maior prioridade
        pq.pop();                      // Remove o elemento de maior prioridade
    }
}

// Saída: 0 1 4 1234
```

pop()

`void pop()`: Remove o elemento de maior prioridade da fila de prioridade. A estrutura da *heap* é ajustada após a remoção para garantir que o próximo elemento de maior prioridade esteja no topo.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq(12);  // Cria uma fila de prioridade com capacidade 12

    pq.push(1);     // Insere o elemento 1 na fila
    pq.push(1234);  // Insere o elemento 1234 na fila
    pq.push(4);     // Insere o elemento 4 na fila
    pq.push(0);     // Insere o elemento 0 na fila
```

```
    std::cout << pq.top() << " ";

    pq.pop();  // Remove o elemento de maior prioridade da fila

    std::cout << pq.top() << " ";
}

// Saída: 0 1
```

top()

`Type top()`: Retorna uma cópia para o elemento de maior prioridade na fila de prioridade, que é o elemento no topo da *heap*. Caso a fila esteja vazia, esse método lançará uma exceção `std::runtime_error`.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq(12);  // Cria uma fila de prioridade com capacidade 12

    pq.push(56);
    pq.push(189);
    pq.push(24);
    pq.push(2356);
    pq.push(33);

    std::cout << pq.top() << " ";
}

// Saída: 24
```

empty()

`bool empty() const`: Verifica se a fila de prioridade está vazia, retornando `true` se estiver vazia e `false` caso contrário.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

void PQVazia(PriorityQueue<int> &pq) {
    if (pq.empty()) {
        std::cout << "A fila esta vazia" << std::endl;
    } else {
        std::cout << "A fila nao esta vazia" << std::endl;
    }
}
```

```
int main() {
    PriorityQueue<int> pq;
    PQVazia(pq);
    pq.push(10);
    PQVazia(pq);
}

// Saída: A fila esta vazia
//        A fila nao esta vazia
```

size()

`unsigned size() const`: Retorna o número atual de elementos na fila de prioridade, ou seja, o tamanho atual da fila.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq;
    std::cout << pq.size() << " ";
    pq.push(1);
    pq.push(2);
    std::cout << pq.size() << " ";
    pq.push(3);
    pq.push(4);
    pq.push(5);
    std::cout << pq.size() << " ";
    pq.pop();
    std::cout << pq.size() << " ";
}

// Saída: 0 2 5 4
```

print()

`void print()`: Este método imprime os elementos da fila de prioridade, na ordem que estão no vetor interno. Se a fila estiver vazia, ele lançará uma exceção `std::runtime_error`.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq;
```

```
    pq.push(1234);
    pq.push(55);
    pq.push(456783);
    pq.push(312);
    pq.push(54);

    pq.print();
}

// Saída: 54 55 456783 1234 312
```

Métodos privados

parent()

`int parent(unsigned i)`: Este método é um auxiliar que calcula o índice do pai de um nó na *min heap*. A fórmula utilizada é `(i - 1) / 2`, onde `i` é o índice do nó em questão. Isso é essencial para navegar na estrutura da *heap* durante operações como inserção e correção (*heapify*).

left()

`int left(unsigned i)`: Este método calcula o índice do filho esquerdo de um nó na *min heap*. A fórmula usada é `2 * i + 1`, onde `i` é o índice do nó pai. O índice retornado é o do filho esquerdo, que é necessário para verificar e realizar correções na estrutura da *heap*.

right()

`int right(unsigned i)`: Este método calcula o índice do filho direito de um nó na *min heap*. A fórmula usada é `2 * i + 2`, onde `i` é o índice do nó pai. O índice retornado é o do filho direito, que também é necessário para verificar e realizar correções na estrutura da *heap*.

swap()

`void swap(int* a, int* b)`: O método é responsável por trocar os valores de dois elementos na *heap*. É usado para reorganizar os elementos durante as operações de inserção e correção (*heapify*). Ele recebe dois ponteiros para inteiros como parâmetros e troca os valores apontados por esses ponteiros.

heapify()

`void heapify(unsigned i)`: Este método é crucial para manter a propriedade da *min heap*. Ele garante que, após a remoção de um elemento ou a inserção de um novo

elemento, a estrutura da *heap* seja corrigida para que o elemento de menor valor (na raiz) permaneça lá. O método `heapify` recebe o índice de um nó e verifica se ele precisa ser trocado com seus filhos para manter a propriedade da *min heap*.

reserve()

`void reserve(unsigned new_capacity)`: O método `reserve` é responsável por aumentar a capacidade da fila de prioridade se necessário. Ele verifica se a capacidade atual é menor que a nova capacidade desejada e, em caso afirmativo, aloca um novo vetor com a capacidade desejada e copia os elementos do vetor original para o novo. Isso é útil para garantir que a fila de prioridade possa acomodar mais elementos quando necessário, evitando realocações frequentes e melhorando o desempenho.

Iteradores

begin()

`iteratorPQ begin()`: Retorna um iterador apontando para o primeiro elemento da fila de prioridade. Ele permite que você acesse o elemento de maior prioridade.

end()

`iteratorPQ end()`: Retorna um iterador apontando para uma posição após o último elemento da fila de prioridade. Ele é usado apenas para indicar o final da iteração, pois não necessariamente retornará o elemento de menor prioridade.

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq;

    pq.push(1234);
    pq.push(55);
    pq.push(456783);
    pq.push(312);
    pq.push(54);

    for (auto it = pq.begin(); it != pq.end(); ++it) {
        std::cout << *it << " ";
    }
}

// Saída: 54 55 456783 1234 312
```

Exemplos de implementação

```
#include <iostream>

#include "TAD'S/PriorityQueue.h"

int main() {
    PriorityQueue<int> pq;

    for (int i = 100; pq.size() < 10; i -= 2) {
        pq.push(i);
    }

    while (!pq.empty()) {
        std::cout << pq.top() << " ";
        pq.pop();
    }
}

// Saída: 82 84 86 88 90 92 94 96 98 100
```