Universidade Federal do Ceará - Campus Quixadá
QXD0010 – Estruturas de Dados
Turma 02A – Ciência da Computação
Prof. Atílio Gomes Luiz

Erica de Castro e Kauan Pablo

28/10/2023

Relatório 1: Entrega das Estruturas de Dados

Após clonar o repositório, abra o diretório `src`, compile o arquivo `main_heap.cpp` e o arquivo `main_queue.cpp` para testar a implementação das respectivas estruturas de dados.

Link para o repositório no GitHub

Equipe

- Erica de Castro Silveira, 552460 - GitHub
- Kauan Pablo de Sousa Silva, 556027 -GitHub

01. Fila (Queue)

Uma fila (queue) é uma estrutura de dados que segue o princípio FIFO (First In, First Out), onde o primeiro elemento inserido é o primeiro a ser removido. Ela é composta por nós que armazenam valores e possuem ponteiros para os nós seguintes; a classe `queue` implementa essa estrutura. O princípio FIFO é frequentemente utilizado em sistemas de gerenciamento de processos devido à política "First Come, First Served (FCFS)." Nessa política, os processos são atendidos na ordem em que chegam.

01.1 Métodos Principais

Aqui estão as explicações dos três principais métodos da classe `queue`:

- `void push(const type &elem)`:

  O método `push` insere um elemento no final da fila. Se a fila estiver vazia, ele cria um novo nó e faz os ponteiros para o primeiro e último nó apontarem para ele. Se a fila não estiver vazia, ele cria um novo nó e faz o ponteiro para o último nó apontar para ele. Após a inserção, o método atualiza o tamanho da fila.

```
// Insere um elemento no final da fila. O(1)
void push(const type &elem) {
    if (m_first == nullptr) {
        // Se a fila estiver vazia, criamos um novo nó e fazemos o
        // ponteiro para o primeiro e último nó apontar para ele.
        m_last = m_first = new node<type>(elem, nullptr);
```

```
        } else {
            // Se não, criamos um novo nó e fazemos o
            // ponteiro para o último nó apontar para ele.
            m_last->get_next() = new node<type>(elem, nullptr);
            m_last = m_last->get_next();
        }
        // Atualizamos o tamanho da fila.
        m_size++;
    }
```

- `void pop()`:

  O método `pop` remove o elemento do início da fila, também seguindo o princípio FIFO (First In, First Out). Se a fila estiver vazia, ele lança uma exceção. Caso contrário, ele cria um nó auxiliar para não perder o ponteiro para o primeiro nó e faz o ponteiro para o primeiro nó apontar para o segundo nó. Em seguida, o ponteiro para o próximo nó do nó auxiliar aponta para `nullptr`, e o método desaloca a memória do nó auxiliar. Após a remoção, o método atualiza o tamanho da fila e, se a fila ficar vazia, atualiza o ponteiro para o último nó.

```
    // Remove o elemento do início da fila. O(1)
    void pop() {
        if (m_size == 0) {
            throw std::runtime_error("fila vazia");
        }

        // Criamos um nó auxiliar para não perder o ponteiro para o
        // primeiro nó, e fazemos o ponteiro para o primeiro nó apontar
        // para o segundo nó.
        node<type> *aux = m_first;
        m_first = m_first->get_next();

        // Fazemos o ponteiro para o próximo nó do nó auxiliar apontar
        // para nullptr e desalocamos a memória do nó auxiliar.
        aux->get_next() = nullptr;
        delete aux;

        // Atualizamos o tamanho da fila, e se a fila ficou vazia,
        //  atualizamos o ponteiro para o último nó.
        m_size--;
        if (m_size == 0) {
            m_last = nullptr;
        }
    }
```

- `type &front()`:

  O método `front` retorna o elemento do início da fila, permitindo que você acesse o valor que será removido a seguir. Se a fila estiver vazia, o método lança uma exceção. Esta função é útil para verificar o valor do elemento na frente da fila antes de removê-lo.

```
    // Retorna o elemento do início da fila. O(1)
    type &front() {
        if (m_size == 0) {
            throw std::runtime_error("fila vazia");
        }
        return m_first->get_value();
    }
```

```
// Versão constante da função front. O(1)
const type &front() const {
    if (m_size == 0) {
        throw std::runtime_error("fila vazia");
    }
    return m_first->get_value();
}
```

02. Fila de Prioridade (Min Heap)

Uma fila de prioridade é uma estrutura de dados que permite armazenar elementos associados a prioridades e garantir que os elementos com maior prioridade sejam processados primeiro. Nesse projeto, a fila de prioridade é implementada como uma min heap. Utilizada para simular a política de escalonamento de processos Shortest Job First (SJF), onde o processo com menor tempo de execução é executado primeiro.

A min heap é um tipo de heap binário baseado em árvore, que funciona da seguinte forma: O valor de cada elemento é menor ou igual ao valor de seus filhos. A raiz da árvore é o elemento de menor valor. A min heap é uma estrutura de dados eficiente para implementar uma fila de prioridade porque permite inserir e remover elementos em tempo O(log n) na maioria dos casos.

02.1 Metódos Principais

Aqui estão as explicações dos principais métodos da classe min heap

- `push(type value)`: Insere um elemento na fila de acordo com sua prioridade. A operação de inserção pode envolver a reorganização da árvore (subida na heap) para manter a propriedade da min heap, para isso, utilizamos o método `up`.

```
// Insere um elemento na heap. O(n) no pior caso (capacidade cheia) e
// O(log n) no resto dos casos.
void push(type value) {
    // Insere elemento no final do heap e incrementa o tamanho.
    ptr[m_size] = value;
    m_size++;

    // Dobra a capacidade do vetor se necessário. O(n)
    if (m_size == m_capacity) {
        reserve();
    }

    // Chama a função up para colocar o elemento na posição correta.
    // O(log n)
    up(m_size - 1);
}
```

- `pop()`: Remove o elemento de maior prioridade da fila (menor elemento), que é o elemento na raiz da min heap. Após a remoção, a árvore é reorganizada (descida na heap) utilizando o método `down` para manter a propriedade da min heap.

```
// Remove o elemento da raiz da heap. O(log n)
void pop() {
    // verifica se a heap está vazia, se estiver, lança uma exceção.
    if (m_size == 0) {
        throw std::out_of_range("Heap is empty");
    }
```

```
        // Troca o elemento da raiz com o último elemento da heap e
        // decrementa o tamanho.
        swap(&ptr[0], &ptr[m_size - 1]);
        m_size--;

        // Chama a função down para colocar o novo elemento da raiz na
        // posição correta. O(log n)
        down(0);
    }
```

- `root()`: Retorna o elemento de maior prioridade da fila (menor elemento), que é o elemento na raiz.

```
    // Retorna o elemento da raiz da heap. O(1)
    type root() {
        if (m_size == 0) {
            throw std::out_of_range("Heap is empty");
        }
        return ptr[0];
    }

    // Versão constante da função root. O(1)
    const type root() const {
        if (m_size == 0) {
            throw std::out_of_range("Heap is empty");
        }
        return ptr[0];
    }
```

02.2 Métodos `up` e `down`

Além das operações principais, a min heap requer dois métodos essenciais para manter sua propriedade:

- `up(int index)`: O método up move um elemento para cima na heap, garantindo que o pai seja menor que os filhos. Isso é necessário após a inserção de um elemento. A função é chamada com o índice do elemento recém-inserido e, se necessário, troca o elemento com seu pai e chama recursivamente up para o pai.

```
    // Move um elemento para cima na heap, garantindo que o pai seja
    // menor que os filhos. O(log n)
    void up(int index) {
        // Verifica se o elemento é a raiz, pois se for, não tem pai.
        if (index == 0) {
            return;
        }

        // Calcula o índice do pai.
        int daddy = (index - 1) / 2;

        // Verifica se o pai é maior que seu filho.
        // Se for, troca o pai com o filho e chama a função up para o
        // index do pai, pois o pai pode ser maior que o seu pai.
        if (ptr[daddy] > ptr[index]) {
            swap(&ptr[daddy], &ptr[index]);
```

```
            up(daddy);
        }
    }
```

- `down(int index)`: O método `down` move um elemento para baixo na heap, garantindo que o filho seja maior que o pai. Isso é necessário após a remoção do elemento de maior prioridade. A função é chamada com o índice do elemento removido, e se necessário, troca o elemento com seu filho de menor valor e chama recursivamente `down` para o filho.

```
// Move um elemento para baixo na heap, garantindo que o filho
// seja maior que os pais. O(log n)
void down(int index) {
    // Calcula o índice do filho esquerdo, e assume que o filho
    // esquerdo é o menor filho.
    int index_smaller = (index * 2) + 1;

    // Verifica se existe filho esquerdo.
    if (index_smaller >= this->m_size) {
        return;
    }

    // Verifica se existe o filho direito.
    // Se existir, verifica se o filho direito é menor que o
    // filho esquerdo e atualiza o índice do menor filho.
    if ((index_smaller + 1) < this->m_size) {
        if (ptr[index_smaller] > ptr[index_smaller + 1]) {
            index_smaller += 1;
        }
    }

    // Verifica se o elemento i (pai) é maior que o menor filho.
    // Se for, troca o elemento i com o menor filho e chama a
    // ser maior que os seus função down para o menor filho, pois
    //  o menor filho pode filhos.
    if (ptr[index] > ptr[index_smaller]) {
        swap(&ptr[index], &ptr[index_smaller]);
        down(index_smaller);
    }
}
```

03. Conclusão

Concluímos a primeira parte do projeto, implementando com sucesso as estruturas de dados fila e fila de prioridade. A próxima etapa envolve a implementação dos algoritmos de escalonamento SJF e FCFS, que serão utilizados para simular o escalonamento de processos em vários computadores. Aguarde o relatório da segunda parte em breve