Segundo Trabalho Avaliativo de Estrutura de Dados Básica II

Raoni Silva, Pedro Galvão, Hélio Lima e Thiago Nascimento

January 8, 2025

Turma 35M34
Unidade 2

Contents

1 Ambiente Computacional

Todas as implementações foram executadas e cronometradas no mesmo ambiente computacional, para uma comparação justa e adequada entre as diferentes formas de implementação das funções citadas no roteiro avaliativo.

O ambiente computacional consiste em uma máquina virtual no servidor pessoal de um dos integrantes do grupo. A máquina virtual possui dois núcleos do processador, 2GB de memória RAM e 32GB de disco rígido à sua disposição, rodando uma distribuição Linux conhecida como Arch Linux, com uma insta- lação mínima, contendo somente os serviços necessários para funcionamento do sistema operacional e o serviço de servidor SSH, para conexão remota com a máquina.

Todas as implementações foram cronometradas utilizando os métodos apropriados para cada linguagem de programação e seus respectivos tempos de execução foram armazenados em um arquivo final, para serem analisados no presente relatório.

As implementações da heap foram feitas em Rust, e as árvores foram implementadas em C. Até existiu a tentativa de criar a árvore rubro-negra em rust, mas devido a necessidade de ter referências cíclicas, a implementação em Rust fica muito complicada.

2 Heap

Nessa seção, são explicadas as estruturas de MinHeap e MaxHeap. Ambas foram implementadas em Rust, devido as atividades da primeira unidade, pois as ordenações da primeira únidade foram feitas em Rust. Primeiramente explicaremos a Max Heap e após isso, mais resumidamente devido a semelhança, a Min Heap.

2.1 Max Heap

A Max Heap é uma espécie de árvore, que é comumente implementada como lista. A única condição para que uma árvore/lista seja considerada uma Max Heap, é que para todo nó, seu filhos devem ter prioridade menor que o pai.

Desse modo, nota-se que o maior elemento da Max Heap sempre será a raiz dela (ou o H[0], no caso da lista).

2.1.1 Estrutura

Na implementação da Max Heap, criamos um WrapperType(uma classe), para encapsular o funcionamento da Max Heap:

```
pub struct MaxHeap<T> {
    data: Vec<T>,
}
```

2.1.2 Alteração de prioridade

Para realizar a alteração de prioridade na heap(o tira casaco, bota casaco dela), é preciso implementar as funções de subir e descer na heap. Elas servem para manter a principal propriedade da (max)heap: cada nó tem prioridade maior que seus filhos.

1. Função subir

Para a função de subir(bubble_up), a implementação é simples. Pegamos a heap(&mut self) e a posição que ira subir como argumentos. Devido as propriedades da heap, sabemos que o pai da $self[i]$ está na posição $i/2$, e dessa forma verificamos se o filho tem prioridade maior que o pai. Se for o caso, as posições do filho e do pai são trocadas, e então chama-se a função recursivamente na posição do pai.

```
pub fn bubble_up(&mut self, mut index: usize) {
    while index > 0 {
        let parent = (index - 1) / 2;
        if self[index] <= self[parent] {
            break;
        }
        self.swap(index, parent);
        index = parent;
    }
}
```

2. Função subir

Para a função de descer, é um pouco mais complicado. Visto que cada item da heap terá 2 filhos, é preciso levar em conta os dois, para decidir o que fazer no algoritmo.

Primeiramente, pegamos a quantidade de elementos na Heap e então fazemos uma iteração.

Para cada iteração, comparamos a prioridade do index atual com a prioridade de seus filhos, caso algum dos filhos seja maior que o pai, realizamos o swap do pai com o filho, e repetimos o processo. Se nenhum dos filhos é maior que o pai, significa que o item desceu até a posição correta, e paramos o loop.

```
pub fn bubble_down(&mut self, mut index: usize) {
    let last_index = match self.len() {
        0 => 0,
        n => n - 1,
    };
    loop {
        let left_child = (2 * index) + 1;
        let right_child = (2 * index) + 2;
        let mut largest = index;
        if left_child <= last_index && self[left_child] >
           self[largest] {
            largest = left_child;
        }
        if right_child <= last_index && self[right_child] >
           self[largest] {
            largest = right_child;
        }
        if largest == index {
            break;
        }
        self.swap(index, largest);
        index = largest;
    }
}
```

2.1.3 Inserção

3 Árvore Binária

A implementação da estrutura de árvore binária foi feita em C devido ao controle preciso de memória e à utilização eficiente de ponteiros, características essenciais dessa linguagem.

3.1 Definição

Uma árvore binária é uma estrutura de dados composta por um conjunto finito de elementos, chamados de nós, sendo que o primeiro nó, denominado raiz, é o ponto inicial da árvore e o os nós da base são conhecidos como folhas. Em uma árvore binária, cada nó pode ter no máximo dois filhos, um à esquerda e outro à direita.

A árvore binária tem a propriedade de que todos os nós de uma subárvore à direita de um nó são maiores do que o valor armazenado na raiz desse nó, enquanto todos os nós de uma subárvore à esquerda de um nó são menores do que o valor armazenado na raiz desse nó. Além disso, cada subárvore, formada pelos filhos à esquerda e à direita de qualquer nó, é também uma árvore binária por si mesma. Isso torna a estrutura recursiva, em que cada nó pode ser considerado a raiz de uma nova árvore binária.

Essa organização permite operações eficientes de busca, inserção e remoção, com a garantia de que a árvore estará estruturada de forma hierárquica e balanceada.

3.2 Estrutura

Na implementação da árvore binária, utilizamos uma estrutura do tipo struct para representar os nós da árvore. Cada nó contém três componentes principais: um valor, representado pela chave, e dois ponteiros, um para a subárvore à esquerda e outro para a subárvore à direita. A seguir, temos a definição da estrutura:

```
1 typedef struct arvore_t {
2   int chave;
3   struct arvore_t *esq;
4   struct arvore_t *dir;
5 } arvore_t;
```

3.3 Operações de Manipulação

3.3.1 Inserção

Na operação de inserção em uma árvore binária, é fundamental que as propriedades da árvore sejam preservadas. Isso significa que todos os nós da subárvore à esquerda de um nó devem conter valores menores que a chave desse nó, e todos os nós da subárvore à direita devem conter valores maiores. Além disso, todo novo nó inserido na árvore sempre será uma folha, ou seja, não possuirá filhos imediatamente após sua criação.

A seguir, apresentamos o código que implementa essa operação:

```c
arvore_t *inserir(arvore_t *arvore, int chave) {
  if (arvore == NULL) {
    arvore = (arvore_t *)malloc(sizeof(arvore_t));
    arvore->chave = chave;
    arvore->esq = NULL;
    arvore->dir = NULL;
  } else if (chave < arvore->chave) {
    arvore->esq = inserir(arvore->esq, chave);
  } else if (chave > arvore->chave) {
    arvore->dir = inserir(arvore->dir, chave);
  }
  return arvore;
}
```

3.3.2 Criação de Árvore a partir de uma Lista

Para construir uma árvore binária a partir de uma lista, de forma que ela fique balanceada ou aproximadamente balanceada, foi adotada a seguinte estratégia:

Etapa 1: Início.

Etapa 2: Verifica se o tamanho da lista é 0.

- Se sim, retorna `NULL`.
- Caso contrário, chama a função `construir_arvore`.

Etapa 3: Calcula o índice do meio.

Etapa 4: Insere a chave central na árvore.

Etapa 5: Chama recursivamente:

- Subárvore esquerda: `construir_arvore(chaves, inicio, meio-1, arvore)`.
- Subárvore direita: `construir_arvore(chaves, meio+1, fim, arvore)`.

Etapa 6: Retorna a árvore construída.

Etapa 7: Finaliza liberando memória.

O código a seguir implementa essa abordagem:

```c
arvore_t *construir_arvore(int *chaves, int inicio, int fim,
    arvore_t *arvore) {
  if (inicio > fim) {
    return arvore;
  }
  int meio = (inicio + fim) / 2;
  arvore = inserir(arvore, chaves[meio]);
  arvore = construir_arvore(chaves, inicio, meio - 1, arvore
     ); // Lado esquerdo
  arvore = construir_arvore(chaves, meio + 1, fim, arvore);
        // Lado direito
```

```
  return arvore;
}
arvore_t *lista_p_arvore(int *chaves, int tamanho) {
  arvore_t *arvore = NULL;
  if (tamanho == 0) {
    return arvore;
  }
  arvore = construir_arvore(chaves, 0, tamanho - 1, arvore);
  return arvore;
}
```

3.3.3 Remoção

O processo de remoção considera três casos principais: o nó a ser removido não possui filhos, possui apenas um filho ou possui dois filhos. Quando o nó possui dois filhos, o menor elemento da subárvore direita (ou o maior da subárvore esquerda) substitui o nó removido, mantendo a organização da árvore. Para isso, utilizamos um nó auxiliar.

A seguir, apresentamos a implementação da operação de remoção:

```
arvore_t *remover(arvore_t *arvore, int chave) {
  if (arvore == NULL)
    return NULL;
  if (chave < arvore->chave) {
    arvore->esq = remover(arvore->esq, chave);
  } else if (chave > arvore->chave) {
    arvore->dir = remover(arvore->dir, chave);
  } else {
    if (arvore->esq == NULL) {
      arvore_t *temp = arvore->dir;
      free(arvore);
      return temp;
    } else if (arvore->dir == NULL) {
      arvore_t *temp = arvore->esq;
      free(arvore);
      return temp;
    }
    arvore_t *rightMin = find_min(arvore->dir);
    arvore->chave = rightMin->chave;
    arvore->dir = remover(arvore->dir, rightMin->chave);
  }
  return arvore;
}
```

3.4 Operações de Consulta

3.4.1 Busca

Na operação de busca em uma árvore binária, o valor procurado é comparado recursivamente com a chave do nó atual, começando pela raiz. Se o valor for menor

que a chave, a busca continua na subárvore à esquerda; se for maior, prossegue na subárvore à direita. Esse processo se repete até que o valor seja encontrado ou até alcançar uma folha (nó nulo), indicando que o valor não está presente na árvore.

A implementação da busca é apresentada a seguir:

```
arvore_t *buscar(arvore_t *arvore, int chave) {
  if (arvore == NULL) {
    return 0;
  }
  if (chave < arvore->chave) {
    return buscar(arvore->esq, chave);
  }
  if (chave > arvore->chave) {
    return buscar(arvore->dir, chave);
  }
  if (chave == arvore->chave) {
    return arvore;
  }
  return NULL;
}
```

3.4.2 Consulta Em Ordem

Na consulta em ordem, os nós da árvore binária são visitados seguindo a sequência: filho da esquerda, raiz e, por fim, filho da direita. Esse tipo de travessia resulta em uma listagem dos elementos em ordem crescente, caso a árvore seja uma árvore binária de busca.

```
void mostrarEmOrdem(arvore_t *arvore) {
  if (arvore != NULL) {
    mostrarEmOrdem(arvore->esq);
    printf("  %d", arvore->chave);
    mostrarEmOrdem(arvore->dir);
  }
}
```

3.4.3 Consulta Em Pré Ordem

Na consulta em pré-ordem, os nós são visitados na seguinte sequência: raiz, filho da esquerda e filho da direita. Esse método é frequentemente utilizado para gerar uma cópia da árvore ou para fins de visualização estrutural.

```
void mostrarEmOrdem(arvore_t *arvore) {
  if (arvore != NULL) {
    mostrarEmOrdem(arvore->esq);
    printf("  %d", arvore->chave);
    mostrarEmOrdem(arvore->dir);
  }
}
```

3.4.4 Consulta Em Pós Ordem

Na consulta em pós-ordem, a visita aos nós segue a ordem: filho da esquerda, filho da direita e, por último, a raiz. Esse tipo de travessia é útil em operações que envolvem a remoção ou processamento dos nós em uma sequência de base para topo (por exemplo, liberar memória ou avaliar expressões em árvores de sintaxe).

```
void mostrarPosOrdem(arvore_t *arvore) {
  if (arvore != NULL) {
    mostrarPosOrdem(arvore->esq);
    mostrarPosOrdem(arvore->dir);
    printf("  %d", arvore->chave);
  }
}
```

3.4.5 Consulta Em Nível

Para realizar uma consulta em nível, ou seja, visitar os nós da árvore da raiz até as folhas, seguindo uma ordem horizontal (nível por nível), é possível utilizar ferramentas gráficas que convertem arquivos no formato DOT para imagens, como arquivos PNG. Esse proceso permite obter a seguinte visualização da árvore:

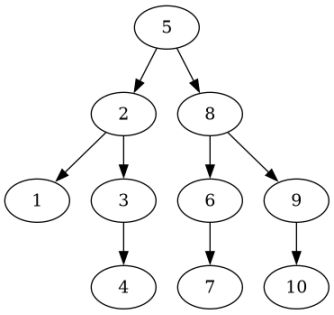

Figure 1: