INSTITUTO FEDERAL
Santa Catarina

Programação 2

HEAP

Objetivos

Objetivo 01

Compreender a estrutura de dados.

Objetivo 02

Demonstrar um pseudocodigo.

Objetivo 03

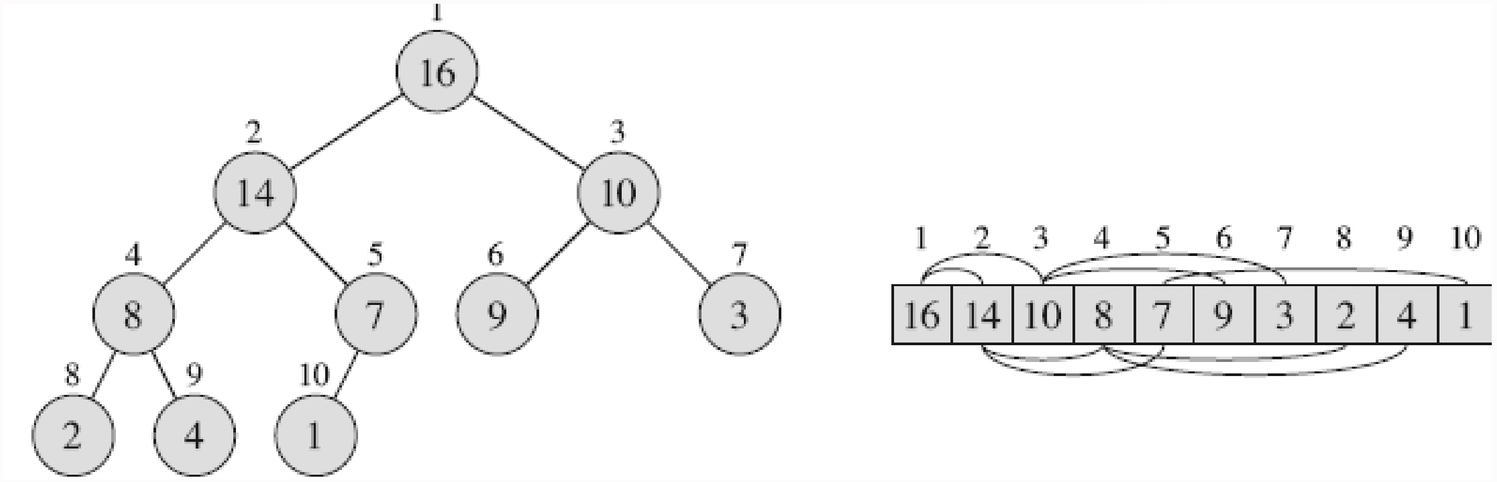
Aplicação do Heap

Objetivo 04

Prós e contras, complexidades em médios e piores casos, é indicado em quais casos?

Compreender a estrutura de dados

- O que é HEAP?
- Como funciona?



Demonstração do pseudocódigo

```
//A função garante que o pai seja maior ou igual a seus filhos.
função alcacao_correta (vetor, n, i):
  maior = i
  valores_esquerda = 2 * i + 1
  valores_direita = 2 * i + 2

//se valores_esquerda é menor que o vetor e valores_esquerda é maior que "maior"
 if valores_esquerda < n e vetor[valores_esquerda] > vetor[valores_esquerda]:
    maior = valores_esquerda

//se valores_direita é menor que o vetor e valores_direita é maior que "maior"
 if valores_direita < n e vetor[valores_direita ] > vetor[valores_direita ]:
    maior = valores_direita

// se maior != i, o indice é trocado
 if maior != i:
    trocar(vetor[i], vetor[largest])
    alcacao_correta(vetor, n, maior)
```

Demonstração do pseudocódigo

```
//a função chama os valores do vetor e começa no indice (n/2) - 1 é
decrementando até o índice 0,
// para alcançar o ultimo pai da arvore binaria
função construir_heap (vetor, n):
  for (int i = (n/2) - 1; i >= 0; i--);
    alcacao_correta(vetor, n, i)

// move o maior elemento do vetor para a posição final
função mover_elemento(vetor, n):
  construir_heap(vetor, n)

//move o maior elemento do vetor auxiliar para o índice atual do vetor original.
  for (int i = n - 1; i >= 0; i--):
    trocar(vetor[0], vetor[i])
    alcacao_correta(vetor, i, 0)
```

Uma aplicação do Heap

```c
//A função garante que o pai seja maior ou igual a seus filhos.
void alcacao_correta(int *vetor, int n, int i) {
    int maior = i;  //é inicializado com o valor de i
    //O índice do filho esquerdo do nó é o dobro do índice do pai, mais 1.
    //filho esquerdo é o primeiro filho do pai.
    int valores_esquerda = 2 * i + 1;
    //filho direiro é o segundo filho do pai.
    int valores_direita = 2 * i + 2;
    //se valores_esquerda é menor que o vetor e valores_esquerda é maior que "maior"
    if (valores_esquerda < n && vetor[valores_esquerda] > vetor[maior]) {
        //atualiza o valor valores_esquerda
        maior = valores_esquerda;
    }
    //se valores_direita é menor que o vetor e valores_direita é maior que "maior"
    if (valores_direita < n && vetor[valores_direita] > vetor[maior]) {
        //atualiza  o valor valores_direita
        maior = valores_direita;
    }
    // se maior != i, o indice é trocado
    if (maior != i) {
        int temp = vetor[i];
        vetor[i] = vetor[maior];
        vetor[maior] = temp;
    //chama alcacao_correta, com o indice maior como argumento
        alcacao_correta(vetor, n, maior);
    }
```

```c
//a função chama os valores do vetor e começa no indice (n/2) - 1 é decrementando até o índice 0
// para alcançar o ultimo pai da arvore binaria
void construir_heap(int *vetor, int n) {
    for (int i = (n/2) - 1; i >= 0; i--) {
        alcacao_correta(vetor, n, i);
    }
}
```

```c
// move o maior elemento do vetor para a posição final
void mover_elemento(int *vetor, int n) {
    //aux tem o mesmo tamanho do vetor original.
    int *aux = malloc(n * sizeof(int));
    //até o vetor ficar vazio
    for (int i = 0; i < n; i++) {
        aux[i] = vetor[i];
    }
    //chama a função construir_heap para construir um heap no vetor auxiliar.
    construir_heap(aux, n);

    //move o maior elemento do vetor auxiliar para o índice atual do vetor original.
    for (int i = n - 1; i >= 0; i--) {
        vetor[i] = aux[0];
        aux[0] = aux[i];
    //chama a função alcacao_correta() para garantir que o heap ainda esteja correto.
        alcacao_correta(aux, i, 0);
    }
    //libera a memoria que foi reservada para aux
    free(aux);
}
```

Uma aplicação do Heap

```c
int main() {
    int vetor[] = {10, 5, 2, 1, 6, 8, 7, 9, 11, 88, 65,78 ,79,
                   651, 54, 56, 456, 54, 53, 22, 21, 33, 23, 56, 76, 42, 22, 99, 44, 31, 27, 91, 777};
    int n = sizeof(vetor) / sizeof(vetor[0]);

    mover_elemento(vetor, n);

    //imprimi o heap
    for (int i = 0; i < n; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
    return 0;
}
```

```
C:\Users\jjami\CLionProjects\ava3\cmake-build-debug\ava3.exe
1 2 5 6 7 8 9 10 11 21 22 22 23 27 31 33 42 44 53 54 54 56 56 65 76 78 79 88 91 99 456 651 777

Process finished with exit code 0
```

Prós e contras

- Prós:

1. **Eficiência na inserção e remoção**: As operações de inserção e remoção em um heap têm complexidade de tempo muito boa. Em um heap binário, por exemplo, a inserção e a remoção de elementos têm complexidade O(log n).

2. **Seleção eficiente do elemento mínimo/máximo:** Heap é ideal para encontrar rapidamente o elemento mínimo ou máximo em um conjunto de elementos.

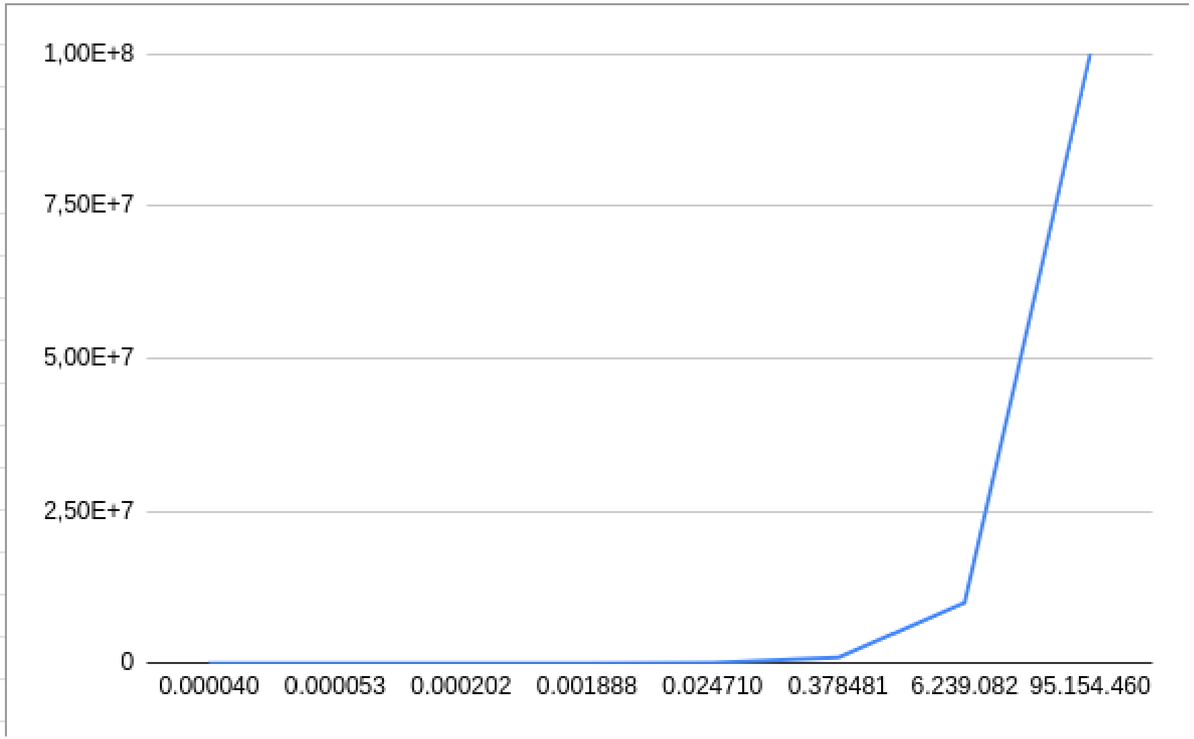
- Contras:

1. Espaço adicional: A estrutura de dados do heap requer exige adicional para armazenar os elementos, pois cada nó da heap precisa armazenar uma chave e um ou dois ponteiros para seus filhos.

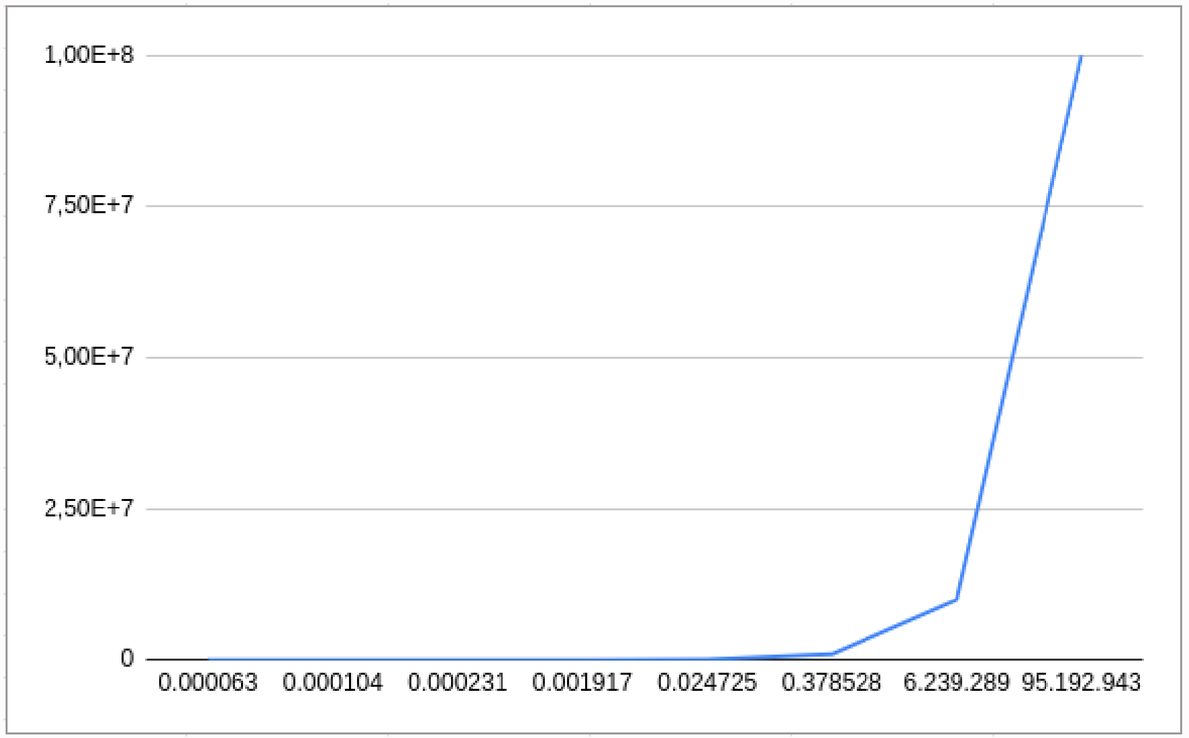
2. Desempenho: O desempenho dos heaps pode ser afetado por fatores como o tamanho da heap e a implementação específica do heap.

Complexidades em médios e piores casos

- Tempo de CPU



- Tempo relógio físico



Complexidade O(n log n)

Complexidades em médios e piores casos

amostras	Tempo gasto de CPU	Tempo gasto
10	0.000040	0.000063
100	0.000053	0.000104
1000	0.000202	0.000231
10000	0.001888	0.001917
100000	0.024710	0.024725
1000000	0.378481	0.378528
10000000	6.239.082	6.239.289
100000000	95.154.460	95.192.943

É indicado em quais casos?

- Ordenação: Heap serve para ordenar uma coleção de dados. O heap é um dos algoritmos de ordenação mais eficientes.
- Prioridade: Os heaps podem ser usados para implementar operações de prioridade, como encontrar o menor ou o maior elemento em uma coleção de dados.
- Alocação de memória: Os heaps podem ser usados para implementar um algoritmo de alocação de memória que aloca memória de forma eficiente.
- Realização de pesquisas: Os heaps podem ser usados para implementar um algoritmo de pesquisa que pesquisa uma coleção de dados de forma eficiente.

Obrigado!