CES-11

Algoritmos e Estruturas de Dados

Carlos Alberto Alonso Sanches
Juliana de Melo Bezerra

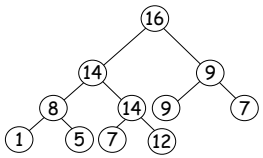
CES-11

- Algoritmos de Ordenação
  - *Heap*
  - Filas de Prioridade
  - *HeapSort*

A estrutura *heap*

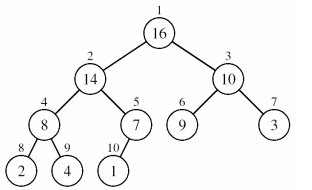
- *Heap* é uma árvore binária com duas propriedades:
  1) Balanceamento: é uma árvore completa, com a eventual exceção do último nível, onde as folhas estão sempre nas posições mais à esquerda.
  2) Estrutural: o valor armazenado em cada nó não é menor que os de seus filhos.

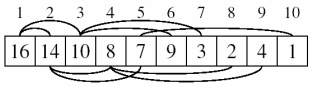
Exemplo:



- *Observação*: Há também o caso análogo, em que o valor de cada nó não é maior que os de seus filhos.

Representação de *heap* com vetor

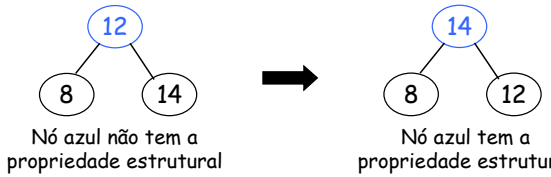




- Armazenamento de um *heap* com n elementos em um vetor v:
  - A raiz está em v[1]
  - O filho esquerdo de v[i] é v[2i]
  - O filho direito de v[i] é v[2i+1]
- O pai de v[i] será v[⌊i/2⌋].
- Os elementos do subvetor v[(⌊n/2⌋+1) .. n] são as folhas.

Propriedade estrutural

- Caso um nó de um *heap* perca a sua propriedade estrutural, poderá recuperá-la trocando de valor com o seu filho maior.
- Isso pode ser feito através do algoritmo *Sift*:



- Como o filho trocado também pode perder a propriedade estrutural, será preciso chamar *Sift* para ele.

Algoritmo *Sift*

Reorganização do *heap* entre as posições i e n:

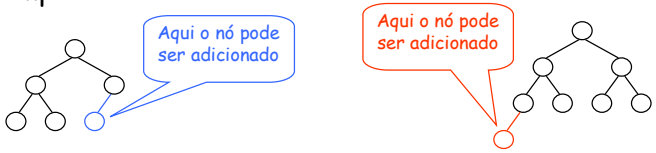
```
Sift(i, n){
    esq = 2i;
    dir = 2i+1;
    maior = i;
    if (esq <= n && v[esq] > v[i])
        maior = esq;
    if (dir <= n && v[dir] > v[maior])
        maior = dir;
    if (maior != i) {
        aux = v[i];
        v[i] = v[maior];
        v[maior] = aux;
        Sift(maior, n);
    }
}
```

Tempo: O(log n)

Exercício: reescrever *Sift* em formato não recursivo.

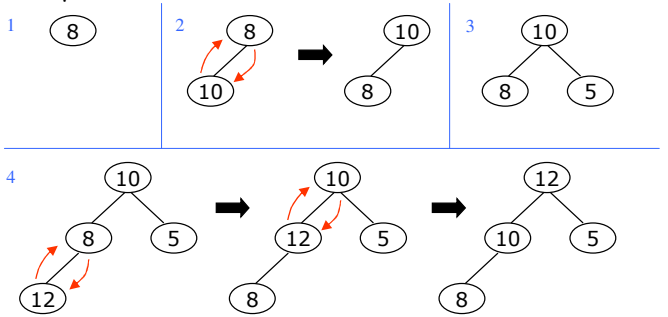
Construção de um *heap*

- Uma árvore com um único nó já é automaticamente um *heap*.
- Procedimento para adição de novos nós a um *heap*:
  - Deverá ser folha no último nível, na primeira posição disponível mais à esquerda.
  - Se este nível estiver cheio, começa-se um novo.
- Exemplos:



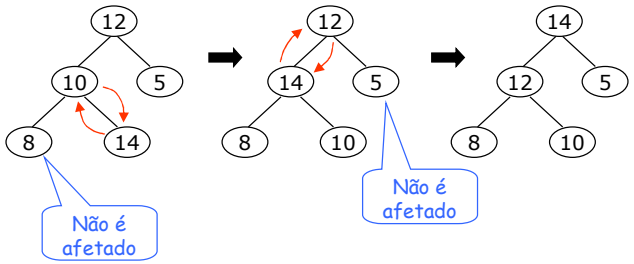
Construção de um *heap*

- Durante o acréscimo de novos nós, se algum nó perder a sua propriedade estrutural, utiliza-se um procedimento análogo ao *Sift*.
- Exemplo:



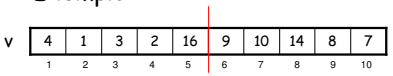
Construção de um *heap*

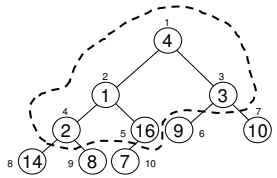
- Essas trocas de valores não afetam a propriedade estrutural dos nós irmãos.
- Exemplo:



Transformação de um vetor em um *heap*

- O algoritmo *Build* transforma o vetor v[1..n] em um *heap* de tamanho n.
- Como as posições entre $\lfloor n/2 \rfloor+1$ e n serão as folhas do *heap*, basta aplicar *Sift* entre as posições 1 e $\lfloor n/2 \rfloor$.
- Exemplo:





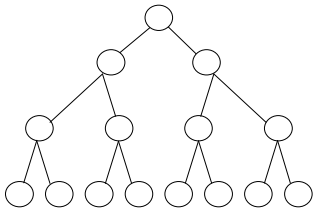

```
Build(v) {
   for (i=⌊n/2⌋; i>0; i--)
        Sift(i, n);
}
```

Complexidade de tempo: O(n.log n) ?

Complexidade de tempo de *Build*

- O tempo gasto por *Sift(i, n)* é proporcional à altura do nó i.
- O pior caso é com a árvore completa:

Altura	Nível	Número de nós
$h = \lfloor \lg n \rfloor$	i = 0	2^0
h - 1	i = 1	2^1
h - 2	i = 2	2^2
h - 3	i = 3	2^3



$$T(n) = 2^0h + 2^1(h-1) + \ldots + 2^h(h-h) \qquad T(n) = \sum_{i=0}^{h} 2^i(h-i)$$

Complexidade de tempo de *Build*

Tempo de pior caso, correspondente a uma árvore completa com n nós e altura h:

$$T(n) = \sum_{i=0}^{h} 2^i(h-i)$$

Multiplicando o numerador e o denominador por 2^h:

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$

Troca de variáveis (k = h - i):

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$

Sabemos que h = lg n e que essa somatória é menor que a correspondente somatória até ∞:

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

Sabemos também que essa somatória é menor que 2:

$$= O(n)$$

CES-11

- Algoritmos de Ordenação
  - *Heap*
  - Filas de Prioridade
  - *HeapSort*

Filas de prioridade

- Fila de prioridade é uma estrutura de dados com as seguintes operações:
  - ***Max*** (ou ***Min***): retorna o valor máximo (ou mínimo) presente na fila
  - ***ExtractMax*** (ou ***ExtractMin***): extrai e retorna o valor máximo (ou mínimo) presente na fila
  - ***Modify(k, x)***: atribui o valor x à posição k da fila
  - ***Insert(x)***: insere o valor x na fila
- *Heap* é uma boa implementação para filas de prioridade.
- Supomos que o *heap* utilize um vetor v, e que a variável `size` armazene o seu tamanho corrente.

Operação *Max*

- Passos:
  - Basta retornar o valor armazenado na primeira posição do *heap*.

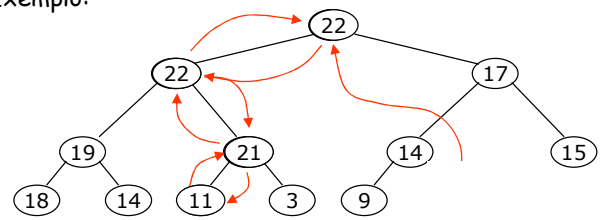
```
Max(){
   return v[1];
}
```

Tempo: *constante*

Um *heap* com operação ***Min*** é análogo.

Remoção da raiz

- Remover a raiz equivale a extrair o máximo valor presente no *heap*.
- Em seguida, colocaremos em seu lugar a última folha e recuperaremos a propriedade estrutural com a aplicação de vários *Sifts*.
- Exemplo:



Operação *ExtractMax*

- Passos:
  1) Substituir a raiz pelo último elemento do *heap*.
  2) Decrementar o seu tamanho atual.
  3) Chamar *Sift* desde a raiz.

```
ExtractMax(){
   if (size < 1)
     Erro("heap underflow");
   else {
      max = v[1];
      v[1] = v[size--];
      Sift(1, size);
      return max;
   }
}
```

Tempo: *logarítmico*

Operação *Modify(k, x)*

- Passos:
  1) Modificar a posição correspondente.
  2) Se a propriedade estrutural for perdida, ir trocando de valor para cima ou para baixo da árvore, até "consertar" o *heap*.

```
Modify(k, x){
  if (k > size || k < 1)
      Erro("Index error");
  else {
     v[k] = x;
     while (k > 1 && v[⌊k/2⌋] < v[k]) { //conserta para cima
           aux = v[k];
           v[k] = v[⌊k/2⌋];
           v[⌊k/2⌋] = aux;
           k = ⌊k/2⌋;
     }
     Sift(k, size); // ou conserta para baixo
  }
}
```

Tempo: *logarítmico*

Operação *Insert(x)*

- Passos:
  1) Aumentar uma posição no final do *heap*, armazenando valor -∞.
  2) Chamar *Modify* nessa posição, alterando para o valor que se deseja inserir.

```
Insert(x){
    v[++size] = −∞;
    Modify(size, x);
}
```

Tempo: *logarítmico*

Sumário

- A tabela abaixo indica as complexidades de tempo das operações de uma fila de prioridades implementada com um *heap*:

Operação	***Tempo***
Build	Linear
Max (ou ***Min***)	Constante
ExtractMax (ou ***ExtractMin***)	Logarítmico
Modify	Logarítmico
Insert	Logarítmico

Exercícios

- Implemente uma fila de prioridades utilizando lista ligada e calcule a complexidade de tempo de cada uma das suas operações.
- Compare esses resultados com a implementação que utiliza *heap*.

CES-11

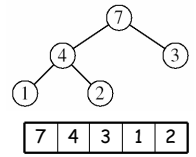
- Algoritmos de Ordenação
  - *Heap*
  - Filas de Prioridade
  - *HeapSort*

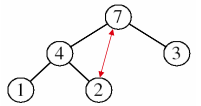
HeapSort (Williams, 1964)

- A estrutura de *heap* permite a elaboração de um eficiente algoritmo de ordenação.
- Ideia:
  1) A partir do vetor inicial, construir um *heap*.
  2) Laço de repetição:
     a) Trocar a raiz (elemento de valor máximo) com o último elemento do *heap*.
     b) Desconsiderar esse último elemento.
     c) Chamar *Sift* para os demais elementos do vetor.

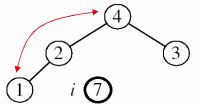
Exemplo

Heap já construído:

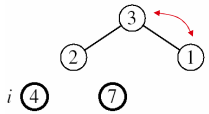




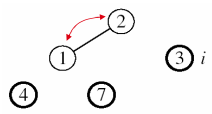
Sift(1, 4)



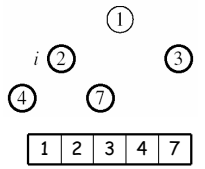
Sift(1, 3)



Sift(1, 2)



Sift(1, 1)



Algoritmo *HeapSort*

```
HeapSort(v) {
    Build(v);
    for (i=n; i>1; i--) {
        aux = v[i];
        v[i] = v[1];
        v[1] = aux;
        Sift(1, i-1);
    }
}
```

- Complexidade de tempo:
  - *Build*: O(n)
  - n-1 *Sifts*: O(n.log n)
- Total: O(n.log n)