Classificação e Pesquisa de Dados

Aula 8

Classificação de dados por Seleção: Seleção Direta e Heapsort

UFRGS INF01124

Classificação por Seleção

Caracteriza-se por identificar, a cada iteração, a chave de menor (maior) valor na porção do vetor ainda não ordenada e colocá-la em sua posição definitiva.

- Principais Algoritmos:
  - Seleção Direta;
  - Heapsort.

Seleção Direta

- Princípio de classificação:
  - A seleção da menor chave é feita por pesquisa seqüencial
  - A menor chave encontrada é permutada com a que ocupa a posição inicial do vetor, que fica reduzido de um elemento
  - O processo de seleção é repetido para o restante do vetor, até que todas as chaves alcancem suas posições definitivas

Qual a diferença para o *bubble sort*?

Exercício

Suponha que se deseja classificar o seguinte vetor utilizando o método da seleção direta:

9 25 10 18 5 7 15 3

Simule as iterações necessárias para a classificação.

[Sugestão de sintaxe para o teste de mesa:
Iteração, conteúdo do vetor, chave, permutação feita, posição atual (ordenados)]

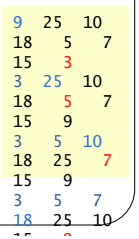
Resolução do Exercício

Iteração Vetor ordenado até a posição	Vetor	Chave Selecionada	Permutação
1	9 25 10 18 5 7 15 3 1	3	9 e 3
2	3 25 10 18 5 7 15 9 2	5	25 e 5
3	3 5 10 18 25 7 15 9 3	7	10 e 7
4	3 5 7 18 25 10 15 9 4	9	18 e 9
5	3 5 7 9 25 10 15 18 5	10	25 e 10
6	3 5 7 9 10 15 25 18 6	18	25 e 18

Ver animação em: http://pt.wikipedia.org/wiki/Selection_sort

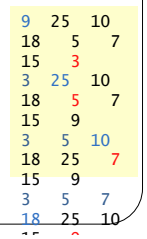
Procedimento de Seleção Direta

```
Proc  seleção direta( c, n ) ;
begin
   for  i ← 1  to  n - 1  do
   begin
      min ← i ;              /* posição do mínimo inicial */
      for  j ← i + 1  to  n  do
      begin
         if   c [ j ]  <  c [ min ]
         then min ← j ;     /* posição do novo mínimo */
      end
      /* troca */
      ch ← c [ i ] ;
      c [ i ] ← c [ min ] ;
      c [ min ] ← ch
   end
end
```



Procedimento de Seleção Direta

```
void selection_sort(int num[], int tam) {
  int i, j, min, troca;
  for (i = 0; i < (tam-1); i++) {
     min = i;
     for (j = (i+1); j < tam; j++) {
       if(num[j] < num[min]) {
          min = j;
       }
     }
     if (i != min) {
        int troca = num[i];
        num[i] = num[min];
        num[min] = troca;
    }
  }
}
```

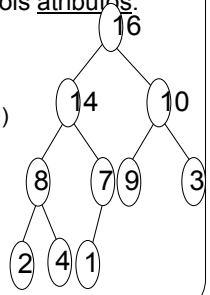


Heapsort

- Autor: J. W. J. Williams, em 1964
- Mesmo princípio:
  - Selecionar o menor item do vetor e trocá-lo com o item que está na primeira posição
  - Repetir essas operações com os n-1 items restantes, n-2, etc.
- Utiliza uma estrutura de dados denominada de heap para organizar a informação durante a execução do algoritmo

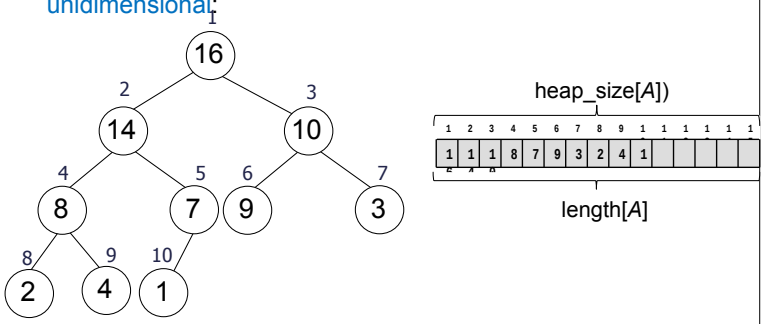
Heap

- **Heap**: vetor que pode ser visto como uma árvore binária totalmente preenchida em todos os níveis exceto, possivelmente, o último
- O último nível é preenchido da esquerda para a direita até um certo ponto
- O vetor *A* que representa um heap possui dois atributos:
  - tamanho (length[*A*]); e
  - número de elementos no vetor (heap_size[*A*])



Heap

- Um heap é visto como uma árvore binária ou como um array unidimensional:



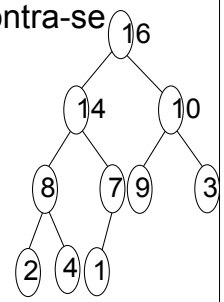
  - Operações de consulta sobre um nodo i:
    - Pai(i): **return**(⌊i/2⌋) → ⌊x⌋ é piso (*floor*) (arrendondamento para baixo)
    - Esquerda(i) **return**(2*i)
    - Direita(i): **return**(2*i + 1)

Propriedade do Heap

- O valor de um nodo é sempre menor ou igual ao valor de seu nodo pai:

  $A[Pai(i)] \geq A[i], \quad \forall\, i \leq heap\_size[A];$

- O elemento de maior valor encontra-se armazenado na raiz da árvore.

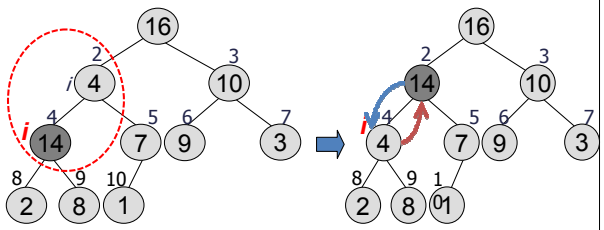


Procedimentos sobre Heaps

- **Heapify**
  - Garante a manutenção da propriedade do Heap (pai ser sempre >=)
- **Build-Heap**
  - Produz um heap a partir de um vetor não ordenado.
  - Usa heapify
- **Heapsort**
  - Procedimento de ordenação local.
  - Usa build-heap
- **Extract-Max e Insert**
  - Permitem utilizar um heap para implementar uma fila de prioridades.

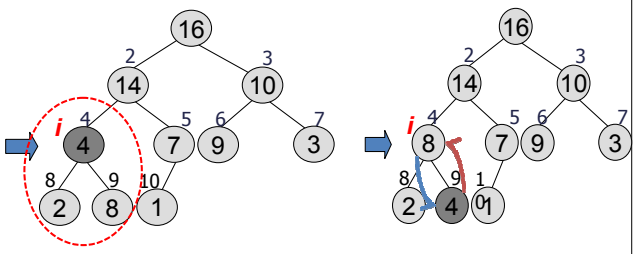
Procedimento Heapify

- Reorganiza heaps
- Supõe que as árvores binárias correspondentes a *Esquerda(i)* e *Direita(i)* são heaps, mas *A[i]* pode ser menor que seus filhos
- Exemplo:



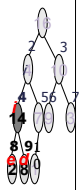
Procedimento Heapify

- Reorganiza heaps
- Supõe que as árvores binárias correspondentes a *Esquerda(i)* e *Direita(i)* são heaps, mas *A[i]* pode ser menor que seus filhos
- Exemplo:



Procedimento Heapify

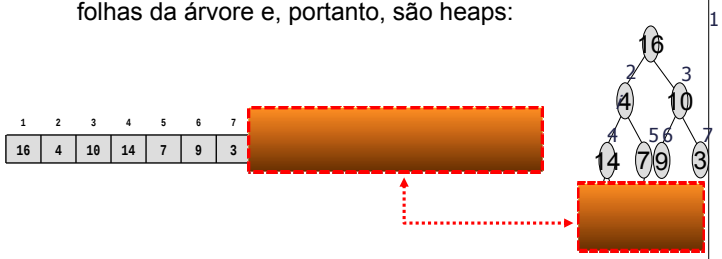
```
Proc  heapify ( A, i )
begin
   e ← Esquerda(i);
   d ← Direita(i);
   maior ← i;
   if  (e ≤ heap_size[A] and A[e] > A[maior]) then
       maior ← e;      /* filho da esquerda é maior */
   if  (d ≤ heap_size[A] and A[d] > A[maior]) then
       maior ← d;      /* filho da direita é maior */
   if  (maior ≠ i) then
      begin
          exchange(A[i] ↔ A[maior]);
          heapify(A, maior);
      end
end
```



Custo: $O(log_2 n)$ – cada troca tem custo $\Theta(1)$ e ocorrem no máximo $log_2 n$ trocas

Procedimento Build-Heap

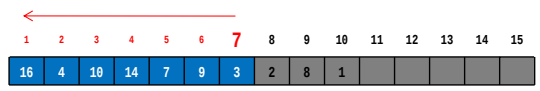
- Os elementos A[(⌊n/2⌋+1)] até A[n] correspondem às folhas da árvore e, portanto, são heaps:

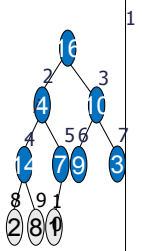


- Logo, basta chamar Heapify para os demais elementos do vetor *A* !
- Build-Heap constroi um Heap de forma bottom-up, chamando o procedimento heapify para transformar um vetor *A[1..n]* em um heap com *n* elementos
- ~~Ao final, o maior elemento estará na primeira posição!~~

Procedimento Build-Heap

```
Proc  build-heap ( A )
begin
  heap_size[A] ← length[A];
  for  i ← ⌊length[A]/2⌋ downto 1 do
     heapify(A, i);
end
```





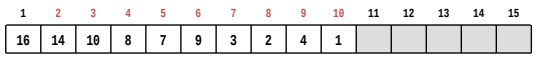
Teste com a árvore ao lado!

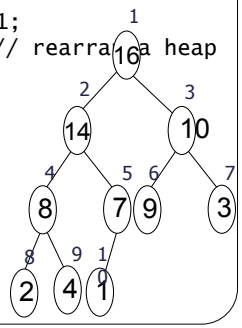
Procedimento Heapsort

- Constrói um heap a partir de um vetor de entrada.
- Como o maior elemento está localizado na raiz (*A[1]*), este pode ser colocado em sua posição final, trocando-o pelo elemento *A[n]*.
- Reduz o tamanho do heap de uma unidade, chama Heapify(A,1) e repete o passo anterior até que o heap tenha tamanho = 2.

Procedimento Heapsort

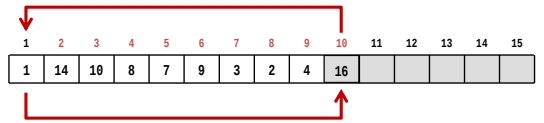
```
Proc  heapsort (A)
begin
(a)  build_heap(A);  // Cria heap e coloca o maior na 1ª posição
   for  i ← length[A] downto 2 do
   begin
        exchange(A[i] ↔ A[1]);       // coloca maior no final
        heap_size[A] ← heap_size[A] -1;
        heapify(A,1);                // rearranja heap
   end
end
```

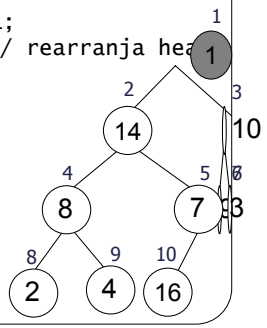




Procedimento Heapsort

```
Proc  heapsort (A)
begin
(a)  build_heap(A);  // Cria heap e coloca o maior na 1ª posição
   for  i ← length[A] downto 2 do
(b) begin
(c)      exchange(A[i] ↔ A[1]);       // coloca maior no final
        heap_size[A] ← heap_size[A] -1;
        heapify(A,1);                // rearranja hea
   end
end
```

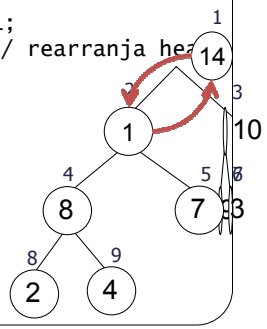




Procedimento Heapsort

```
Proc  heapsort (A)
begin
(a)   build_heap(A);  // Cria heap e coloca o maior na 1ª
posição
      for  i ← length[A] downto 2 do
(b)   begin
(c)         exchange(A[i] ↔ A[1]);       // coloca maior
(d)no final
            heap_size[A] ← heap_size[A] -1;
            heapify(A,1);                // rearranja hea
      end
end
```

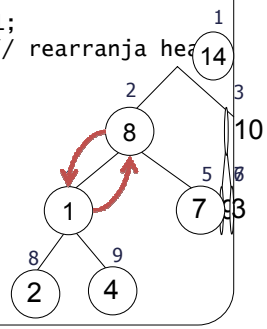
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	1	10	8	7	9	3	2	4	16					



Procedimento Heapsort

```
Proc  heapsort (A)
begin
(a)   build_heap(A);  // Cria heap e coloca o maior na 1ª
posição
      for  i ← length[A] downto 2 do
(b)   begin
(c)         exchange(A[i] ↔ A[1]);       // coloca maior
(d)no final
            heap_size[A] ← heap_size[A] -1;
            heapify(A,1);                // rearranja hea
      end
end
```

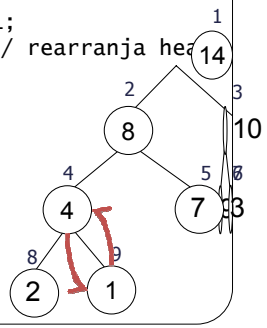
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	8	10	1	7	9	3	2	4	16					



Procedimento Heapsort

```
Proc  heapsort (A)
begin
(a)   build_heap(A);  // Cria heap e coloca o maior na 1ª
posição
      for  i ← length[A] downto 2 do
(b)   begin
(c)         exchange(A[i] ↔ A[1]);       // coloca maior
(d)no final
            heap_size[A] ← heap_size[A] -1;
            heapify(A,1);                // rearranja hea
      end
end
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	8	10	4	7	9	3	2	1	16					



Procedimento Heapsort

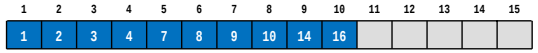
```
Proc  heapsort (A)
begin
      build_heap(A);  // Cria heap e coloca o maior na 1ª
posição
      for  i ← length[A] downto 2 do
(e)*  begin
            exchange(A[i] ↔ A[1]);       // coloca maior
no final
            heap_size[A] ← heap_size[A] -1;
            heapify(A,1);                // rearranja hea
      end
end
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	3	4	7	8	9	10	14	16					

* Ao terminar o laço, todos estão no lugar certo!

Procedimento Heapsort

```
Proc  heapsort (A)
begin
   build_heap(A);  // Cria heap e coloca o maior na 1ª
posição
   for  i ← length[A] downto 2 do
   begin
        exchange(A[i] ↔ A[1]);       // coloca maior
no final
        heap_size[A] ← heap_size[A] -1;
        heapify(A,1);                // rearranja heap
   end
end
```

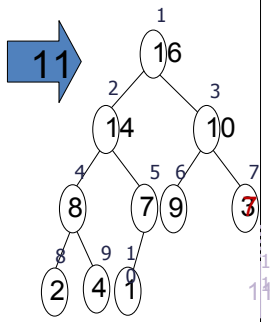


Uso de Heap

- **Fila de Prioridades**
  - Estrutura de dados para manutenção de um conjunto S com *n* elementos, tendo cada elemento uma chave
  - A chave reflete a habilitade relativa do item abandonar o conjunto rapidamente
  - O elemento mais velho (ou a maior chave), sai primeiro
  - Exemplos:
    - Sistemas Operacionais: chave = tempo em que eventos podem ocorrer
    - Gerencia de memória: páginas menos utilizadas são substituídas
  - Suporta as seguintes operações:
    - **Insere_Heap(S,x):** Insere o elemento *x* no conjunto *S*
    - **Máximo(S):** Retorna o elemento de S com maior valor de chave
    - **Extrai_Max(S):** Remove de S e retorna o elemento com o maior valor de chave

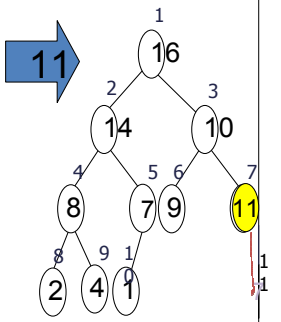
Procedimento Insere_Heap

```
Proc  insere_heap (A, chave)
begin
   heap_size[A] ← heap_size[A] +1;
(a) i ← heap_size[A];
(b) while (i > 1 and A[Pai(i)] < chave) do
   begin
      A[i] ← A[Pai(i)];
      i ← Pai(i);
   end
   A[i] ← chave;
end
```



Procedimento Insere_Heap

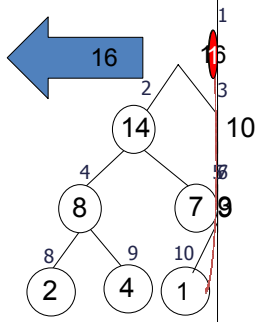
```
Proc  insere_heap (A, chave)
begin
   heap_size[A] ← heap_size[A] +1;
(a) i ← heap_size[A];
(b) while (i > 1 and A[Pai(i)] < chave) do
   begin
(c)    A[i] ← A[Pai(i)];
      i ← Pai(i);
   end
(d) A[i] ← chave;
end
```



Custo: $O(log_2 n)$

Procedimento Extrai_Max

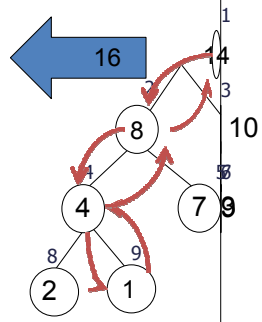
```
Proc  extrai_max (A)
begin
   if (heap_size[A] < 1) then
       error ("heap underflow");
   max ← A[1];
(a) A[1] ← A[heap_size[A]];
(b) heap_size[A] ← heap_size[A] - 1;
   heapify(A,1);
   return (max);
end
```



Procedimento Extrai_Max

```
Proc  extrai_max (A)
begin
   if (heap_size[A] < 1) then
       error ("heap underflow");
   max ← A[1];
(a) A[1] ← A[heap_size[A]];
(b) heap_size[A] ← heap_size[A] - 1;
(c) heapify(A,1);
   return (max);
end
```

Custo: $O(log_2 n)$



Leitura recomendada

- Binary Tree – Wikipedia definition
  http://en.wikipedia.org/wiki/Binary_tree
- Capítulo de HeapSort no livro do prof. Azeredo, P.
  Métodos de Classificação de Dados (ver bibliografia da disciplina).
- Selection Sort no NIST Dictionary of Algorithms and Data Structures
  http://www.itl.nist.gov/div897/sqg/dads/HTML/selectionsrt.html