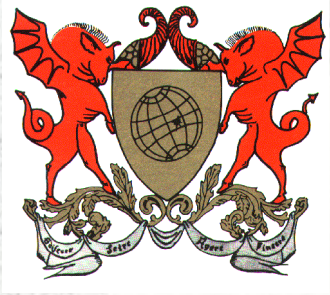
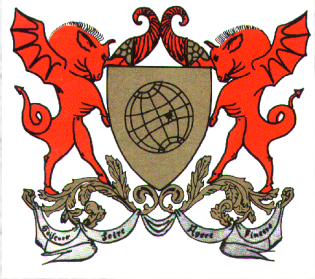


Universidade Federal de Viçosa - Campus Florestal

Algoritmos e Estruturas de Dados I (CCF 211)

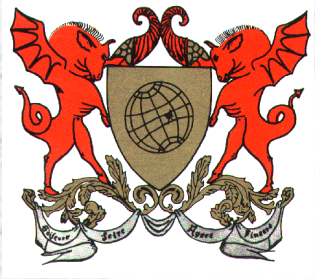
HeapSort

Profa. Thais R. M. Braga Silva
<thais.braga@ufv.br>



Filas de Prioridades

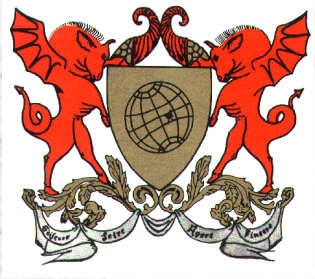
- É uma estrutura de dados onde a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de itens rapidamente.
- Aplicações:
  - SOs usam filas de prioridades, nas quais as chaves representam o tempo em que eventos devem ocorrer.
  - Métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor.
  - Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.



Filas de Prioridades
Tipos Abstratos de Dados

Operações:

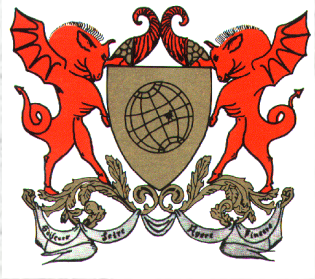
1. Constrói uma fila de prioridades a partir de um conjunto com n itens.
2. Informa qual é o maior item do conjunto.
3. Retira o item com maior chave.
4. Insere um novo item.
5. Aumenta o valor da chave do item i para um novo valor que é maior que o valor atual da chave.
6. Substitui o maior item por um novo item, a não ser que o novo item seja maior.
7. Altera a prioridade de um item.
8. Remove um item qualquer.
9. Agrupar duas filas de prioridades em uma única.



Filas de Prioridade - Representação

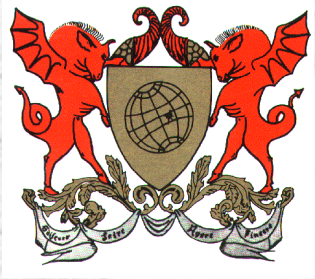
- Lista linear ordenada:
  - Constrói é O(n log n) (ou $O(n^2)$).
  - Insere é O(n).
  - Retira é O(1).
  - Altera é O(n).

- Lista linear não ordenada:
  - Constrói é O(n).
  - Insere é O(1).
  - Retira é O(n).
  - Altera é O(n)



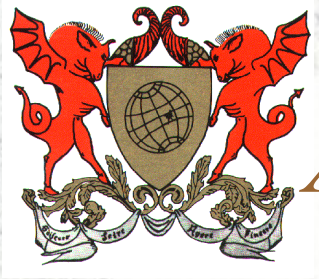
Filas de Prioridade - Representação

- A melhor representação é através de uma estruturas de dados chamada *heap:*
  - Neste caso, Constrói é O(n).
  - Insere, Retira, Substitui e Altera são O(log n).
- **Observação:**
  - Para implementar a operação Agrupar de forma eficiente e ainda preservar um custo logarítmico para as operações Insere, Retira, Substitui e Altera é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais (Vuillemin, 1978).



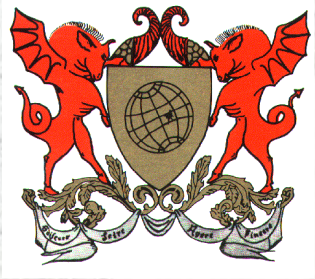
Filas de Prioridades

- As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
- Basta utilizar repetidamente a operação Insere para construir a fila de prioridades.
- Em seguida, utilizar repetidamente a operação Retira para receber os itens na ordem reversa.



Algoritmos de Ordenação Usando Listas de Prioridades

- O uso de listas lineares não ordenadas corresponde ao método ... da seleção
- O uso de listas lineares ordenadas corresponde ao método ... da inserção
- O uso de *heaps* corresponde ao método *Heapsort*.



Heaps

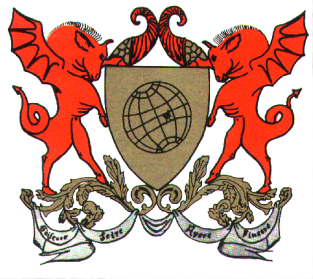
É uma sequência de itens com chaves

c[1], c[2], ... , c[n], tal que:

c[i] >= c[2*i],

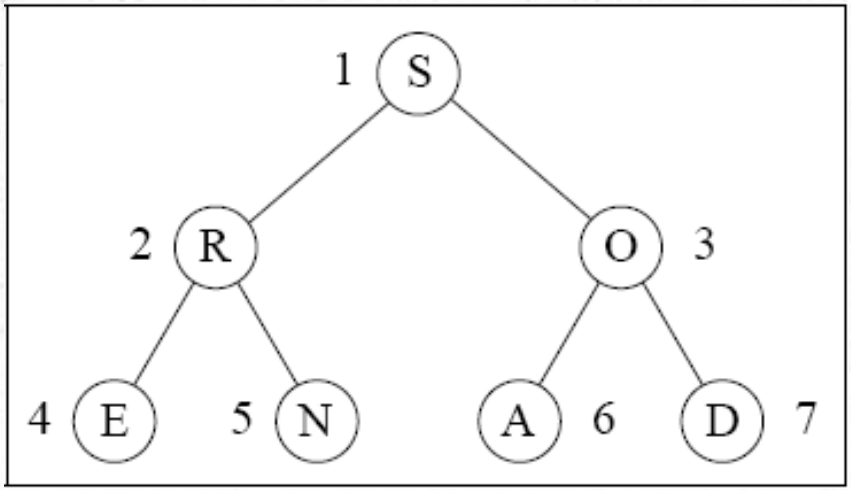
c[i] >= c[2*i + 1],

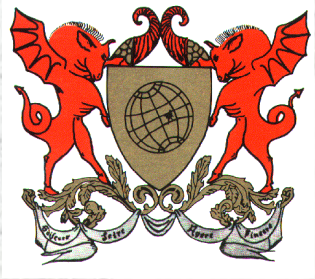
para todo i = 1, 2, ..., n/2



Heaps

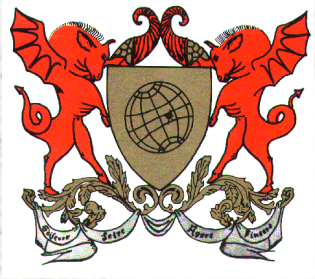
A definição pode ser facilmente visualizada em uma árvore binária completa:





Heaps

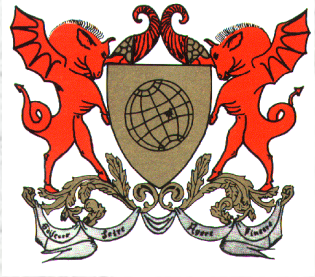
- Árvore binária completa:
  - Os nós são numerados de 1 a n.
  - O primeiro nó é chamado raiz.
  - O nó [k/2] é o pai do nó k, para 1 < k <= n.
  - Os nós 2k e 2k + 1 são os filhos à esquerda e à direita do nó k, para 1 <= k <= [k/2].



Heaps

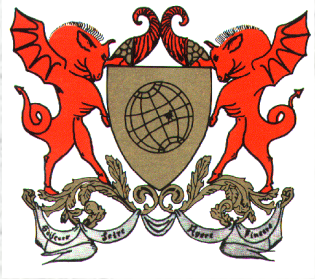
- As chaves na árvore satisfazem a condição do *heap*.
- A chave em cada nó é maior do que as chaves em seus filhos.
- A chave no nó raiz é a maior chave do conjunto.
- Uma árvore binária completa pode ser representada por um vetor:

1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*



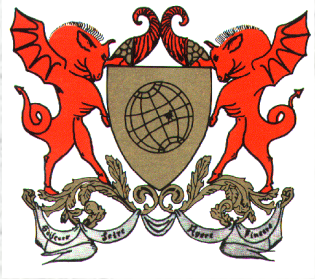
Heaps

- A representação é extremamente compacta.
- Permite caminhar pelos nós da árvore facilmente.
- Os filhos de um nó i estão nas posições 2i e 2i + 1.
- O pai de um nó i está na posição i/2 (i div 2).
- Na representação do *heap* em um arranjo, a maior chave está sempre na posição 1 do vetor.
- Os algoritmos para implementar as operações sobre o *heap* operam ao longo de um dos caminhos da árvore.
- Um algoritmo elegante para construir o *heap* foi proposto por Floyd em 1964.



Heaps

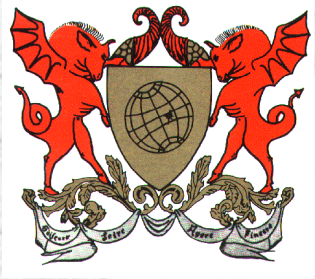
- O algoritmo não necessita de nenhuma memória auxiliar.
- Dado um vetor A[1], A[2], ..., A[n].
- Os itens A[n/2 + 1], A[n/2 + 2], ..., A[n] formam um *heap:*
  - Neste intervalo não existem dois índices i e j tais que j = 2i ou j = 2i + 1.



Heaps

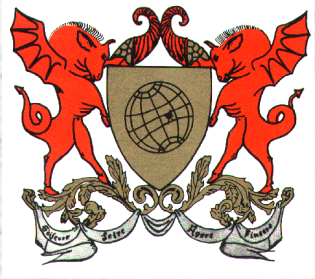
- Algoritmo:

	1	2	3	4	5	6	7
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*	*S*
Esq = 3	*O*	*R*	***S***	*E*	*N*	*A*	***D***
Esq = 2	*O*	*R*	*S*	*E*	*N*	*A*	*D*
Esq = 1	***S***	*R*	***O***	*E*	*N*	*A*	*D*



Heaps

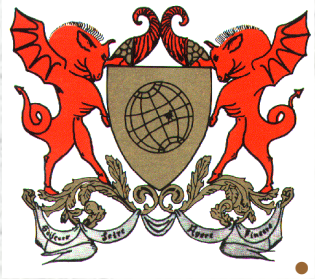
- Os itens de A[4] a A[7] formam um *heap*.
- O *heap* é estendido para a esquerda (Esq = 3), englobando o item A[3], pai dos itens A[6] e A[7].
- A condição de *heap* é violada:
  - *O heap é refeito trocando os itens D e S.*
- O item R é incluído no *heap* (Esq = 2), o que não viola a condição de *heap*.
- O item O é incluindo no *heap* (Esq = 1).
- A Condição de *heap* é violada:
  - *O heap é refeito trocando os itens O e S, encerrando o processo.*



Heaps

- O Programa que implementa a operação que informa o item com maior chave:

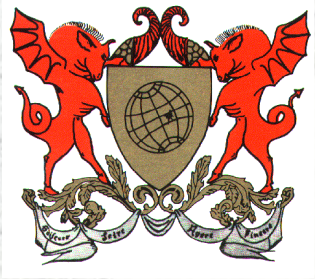
```
Item Max(Item *A)
{
  return (A[1]);
}
```



Heaps

- Programa para refazer a condição de *heap:*

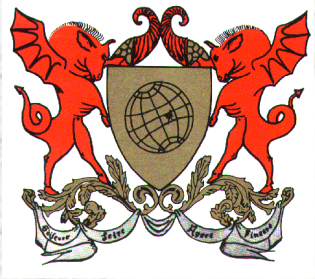
```
void  Refaz(int Esq, int Dir, Item *A)
{
  int i = Esq;
  int j;
  Item aux;
  j = i * 2;
  aux = A[i];
  while (j <= Dir)
  {
    if (j < Dir)
    { if (A[j].Chave < A[j+1].Chave)
        j++; }
    if (aux.Chave >= A[j].Chave)
      break;
    A[i] = A[j];
    i = j; j = i * 2 ;
  }
  A[i] = aux;
}
```



Heaps

- Programa para construir o *heap:*

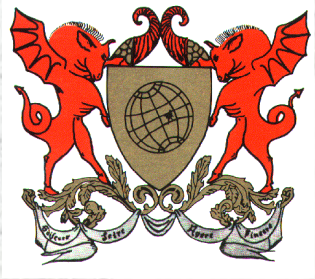
```
void Constroi(Item *A, int *n)
{
  int Esq;
  Esq = *n / 2;
  while (Esq > 0)
  {
    Refaz(Esq, *n, A);
    Esq--;
  }
}
```



Heaps

- Programa que implementa a operação de retirar o item com maior chave:

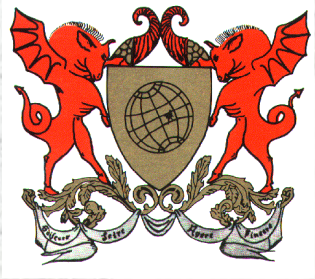
```
int RetiraMax(Item *A, int *n,Item* pMaximo)
{
  if (*n < 1)
    return 0;
  else
  {
    *pMaximo = A[1];
    A[1] = A[*n];
    (*n)--;
    Refaz(1, *n, A);
    return 1;
  }
}
```



Heaps

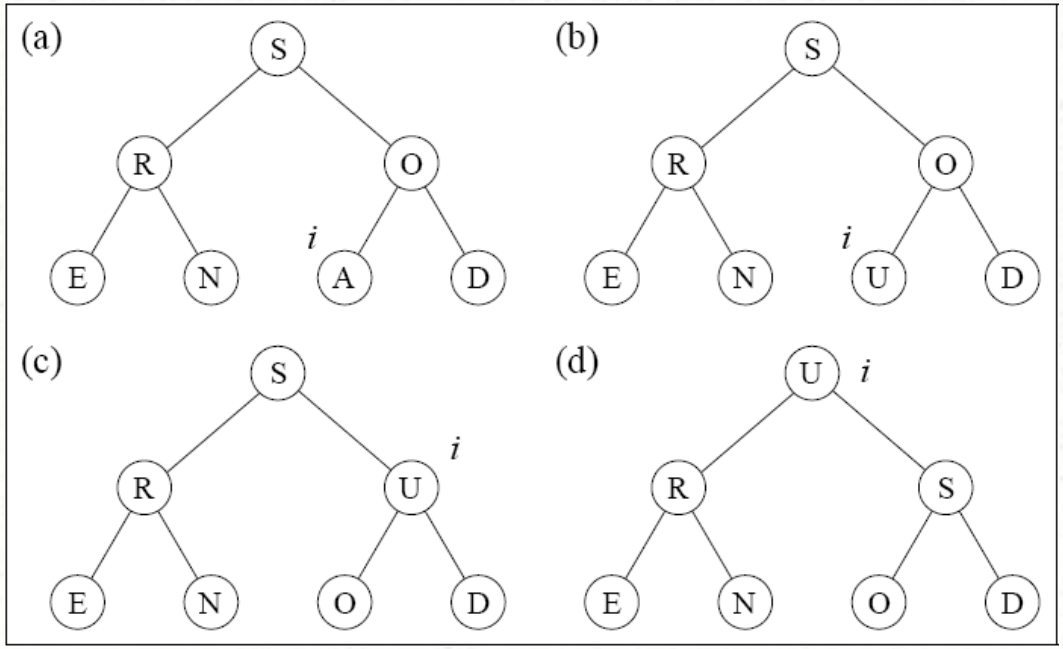
- Programa que implementa a operação de aumentar o valor da chave do item i:

```
int AumentaChave(int i, ChaveTipo ChaveNova, Item *A)
{
  Item aux;
  if (ChaveNova < A[i].Chave)
    return 0;
  A[i].Chave = ChaveNova;
  while (i > 1 && A[i/2].Chave < A[i].Chave)
  {
    aux = A[i/2]; A[i/2] = A[i]; A[i] = aux;
    i /= 2;
  }
  return 1;
}
```

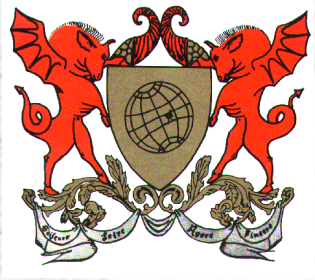


Heaps

- Heaps
  - Exemplo da operação de aumentar o valor da chave do item na posição i:



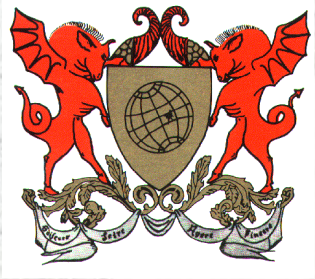
  - O tempo de execução do procedimento AumentaChave em um item do *heap é* O(log n).



Heaps

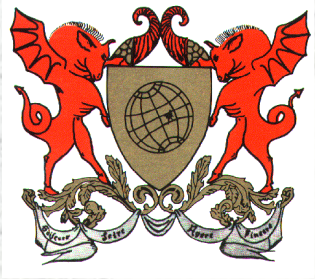
- Programa que implementa a operação de inserir um novo item no *heap:*

```
void Insere(Item *x, Item *A, int *n)
{
   (*n)++;
   A[*n] = *x;
   A[*n].Chave = INT_MIN;
   AumentaChave(*n, x->Chave, A);
}
```



Heapsort

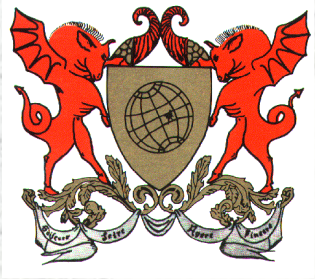
- Algoritmo:
  1. Construir o *heap*.
  2. Troque o item na posição 1 do vetor (raiz do *heap)* com o item da posição n.
  3. Use o procedimento Refaz para reconstituir o *heap* para os itens A[1], A[2], ..., A[n - 1].
  4. Repita os passos 2 e 3 com os n - 1 itens restantes, depois com os n - 2, até que reste apenas um item.



Heapsort

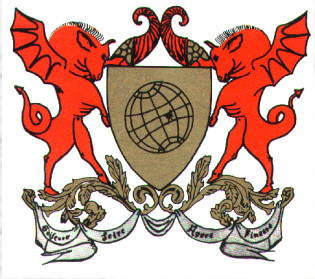
- Exemplo de aplicação do Heapsort:

1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*
R	***N***	*O*	*E*	***D***	*A*	***S***
O	*N*	***A***	*E*	*D*	*R*	
N	***E***	*A*	***D***	*O*		
E	***D***	*A*	*N*			
D	***A***	*E*				
A	*D*					



Heapsort

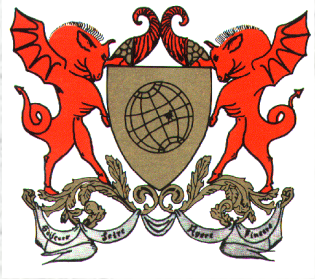
- O caminho seguido pelo procedimento Refaz para reconstituir a condição do *heap* está em negrito.
- Por exemplo, após a troca dos itens S e D na segunda linha da Figura, o item D volta para a posição 5, após passar pelas posições 1 e 2.



Heapsort

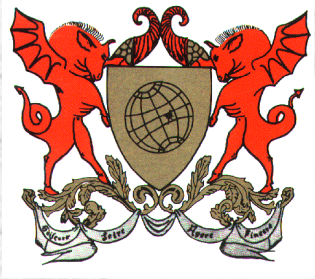
- Programa que mostra a implementação do Heapsort:

```
/* -- Entra aqui a função Refaz -- */
/* -- Entra aqui a função Constroi -- */
void Heapsort(Item *A, Indice *n)
{ Indice Esq, Dir;
  Item aux;
  Constroi(A, n); /* constroi o heap */
  Esq = 1; Dir = *n;
  while (Dir > 1)
  { /* ordena o vetor */
    aux = A[1]; A[1] = A[Dir]; A[Dir] = aux;
    Dir--;
    Refaz(Esq, Dir, A);
  }
}
```



Heapsort

- **Análise**
  - O procedimento Refaz gasta cerca de logn operações, no pior caso.
  - Constrói – executa O(n) x Refaz
  - Laço interno – executa (n) x Refaz
  - Logo, Heapsort gasta um tempo de execução proporcional a nlogn, no pior caso.



Heapsort

- Vantagens:
  - O comportamento do Heapsort é sempre O(nlogn), qualquer que seja a entrada.
- Desvantagens:
  - O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort.
  - O Heapsort não é **estável**.
- Recomendado:
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável.
  - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap*.