Ordenação: Heapsort

Algoritmos e Estruturas de Dados II

Introdução

- Possui o mesmo princípio de funcionamento da ordenação por seleção.
  - Selecione o menor item do vetor
  - Troque-o pelo item da primeira posição
  - Repita operação com os elementos restantes do vetor
- Implementação direta
  - Encontrar o menor elemento requer $n-1$ comparações
- Ideia:
  - Utilização de uma fila de prioridades implementada com um heap.

Fila de Prioridades

- Definição:
  - Estrutura de dados composta de chaves, que suporta duas operações básicas: inserção de um novo item e remoção do item com a maior chave.
  - A chave de cada item reflete a prioridade em que se deve tratar aquele item.

- Aplicações:
  - Sistemas operacionais, sistema de memória (paginação).

Fila de Prioridades

- Operações
  - Constrói a fila de prioridade com N itens
  - Insere um novo item
  - Retira o maior item
  - Altera a prioridade de um item

Fila de Prioridades

- Representações
  - Lista sequencial ordenada, não ordenada e heap.

	Constrói	Insere	Retira máximo
Lista ordenada	O(N log N)	O(N)	O(1)
Lista não ordenada	O(N)	O(1)	O(N)
heaps	O(N log N)	O(log N)	O(log N)

Fila de Prioridades

- Algoritmos de ordenação com fila de prioridades
  - Utiliza operação insere para adicionar todas as N chaves
  - Utiliza a operação retira máximo para receber uma lista na ordem reversa.

	Algoritmo
Lista ordenada	
Lista não ordenada	
heaps	

Fila de Prioridades

- Algoritmos de ordenação com fila de prioridades
  - Utiliza operação insere para adicionar todas as N chaves
  - Utiliza a operação retira máximo para receber uma lista na ordem reversa.

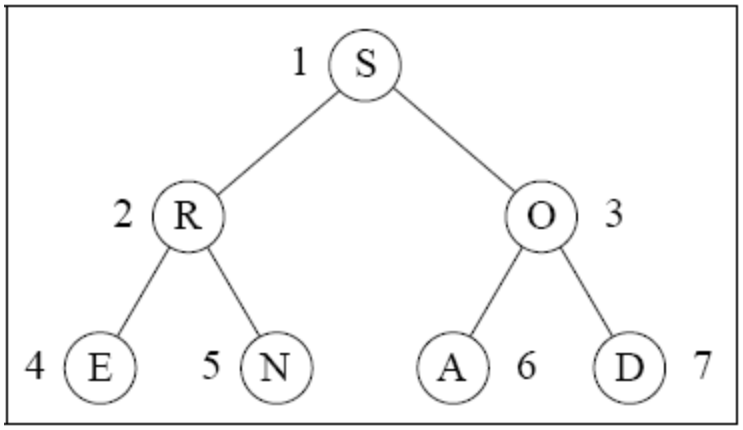
	Algoritmo
Lista ordenada	Inserção
Lista não ordenada	Seleção
heaps	Heapsort

Heap

- Representação vetorial A[1], A[2], ..., A[n]
  - Será um heap se A[i] ≥A[2i] e A[i] ≥ A[2i+1] para todo i = 1, 2, 3, ..., n/2.

- Representação de árvore binária
  - Será um heap se cada nó for maior ou igual seus filhos.

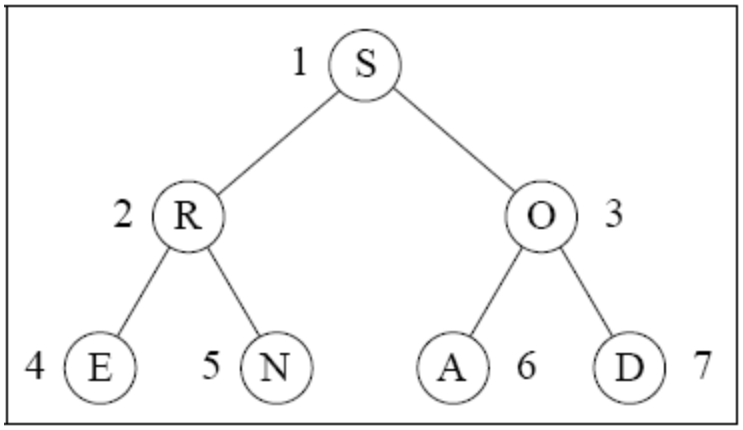
1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*



Heap

- Representação vetorial para de árvore
  - Nós são numerados de 1 a n
  - O primeiro é chamado raiz
  - O nó k/2 é o pai do nó k, $1 < k \leq n$
  - Os nós 2k e 2k+1 são filhos da esquerda e direita do nó k, para $1 \leq k \leq n/2$.

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

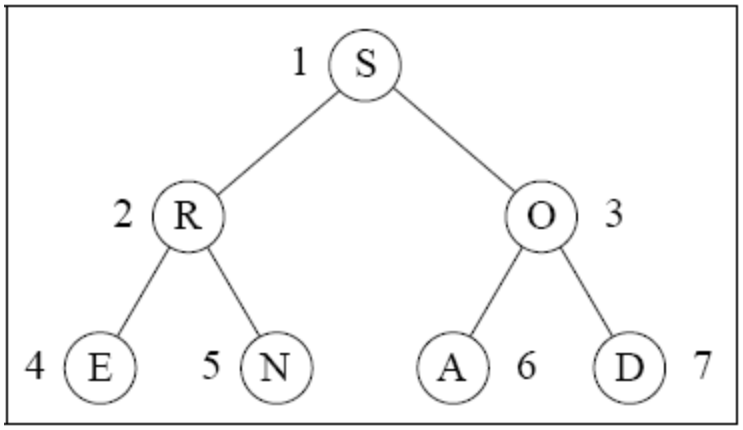


Heap

- Representação por meio de vetores é compacta

- Permite caminhar pelos nós da árvore facilmente
  - Filhos de um nó i estão nas posições $2i$ e $2i + 1$
  - O pai de um nó i está na posição $i/2$
  - A maior chave sempre está na posição 1

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



Heap – adição de elemento no fim

- Restauração da condição de heap (adição no fim)
  - Garantir que o valor da chave do pai é maior que dos filhos.

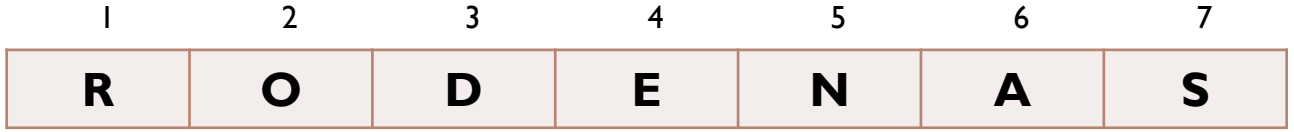
1	2	3	4	5	6	7
R	O	D	E	N	A	S

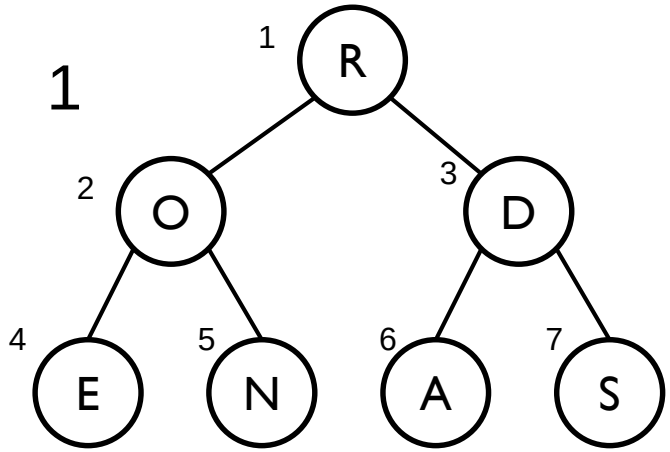
  - Tenda se todos os elementos A[2i] e A[2i + 1] são menores ou igual a A[i].

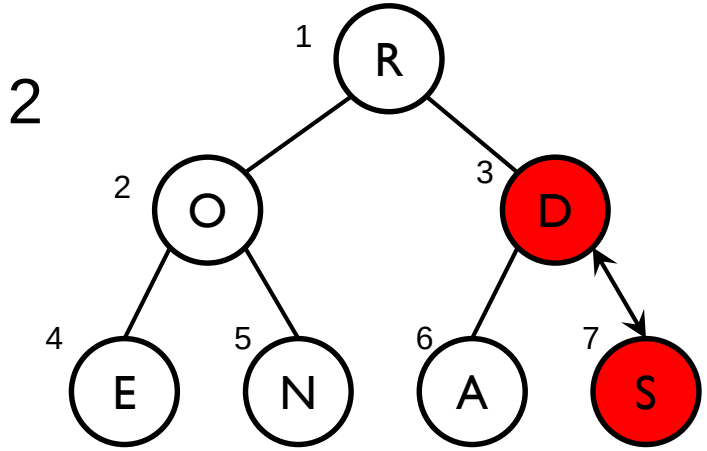
1	2	3	4	5	6	7
R	O	D	E	N	A	S
R	O	S	E	N	A	D
S	O	R	E	N	A	D

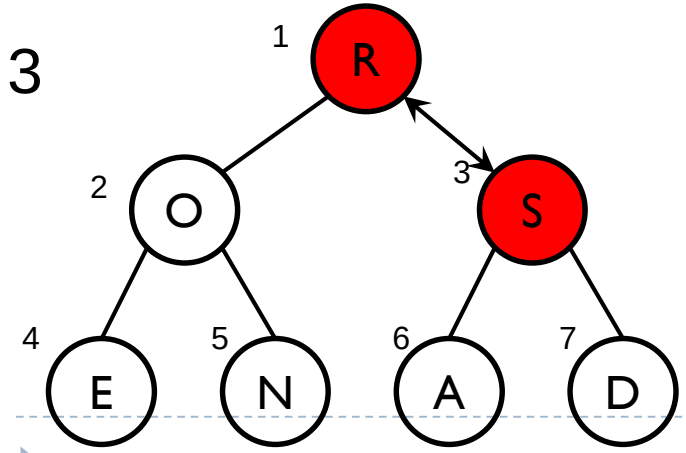
Heap – adição de elemento no fim

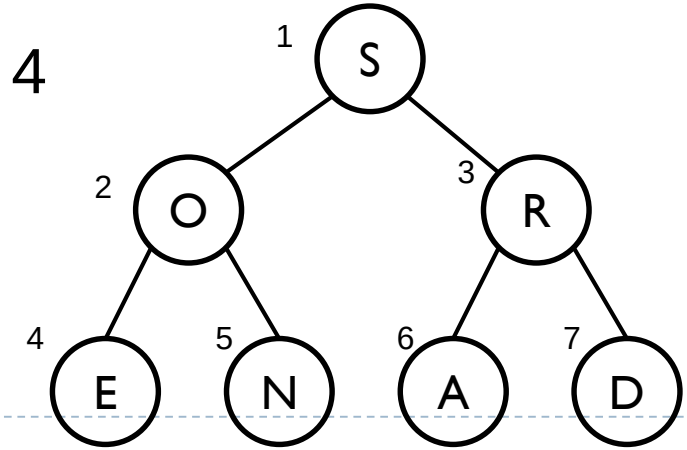
- Restauração da condição de heap (árvore)









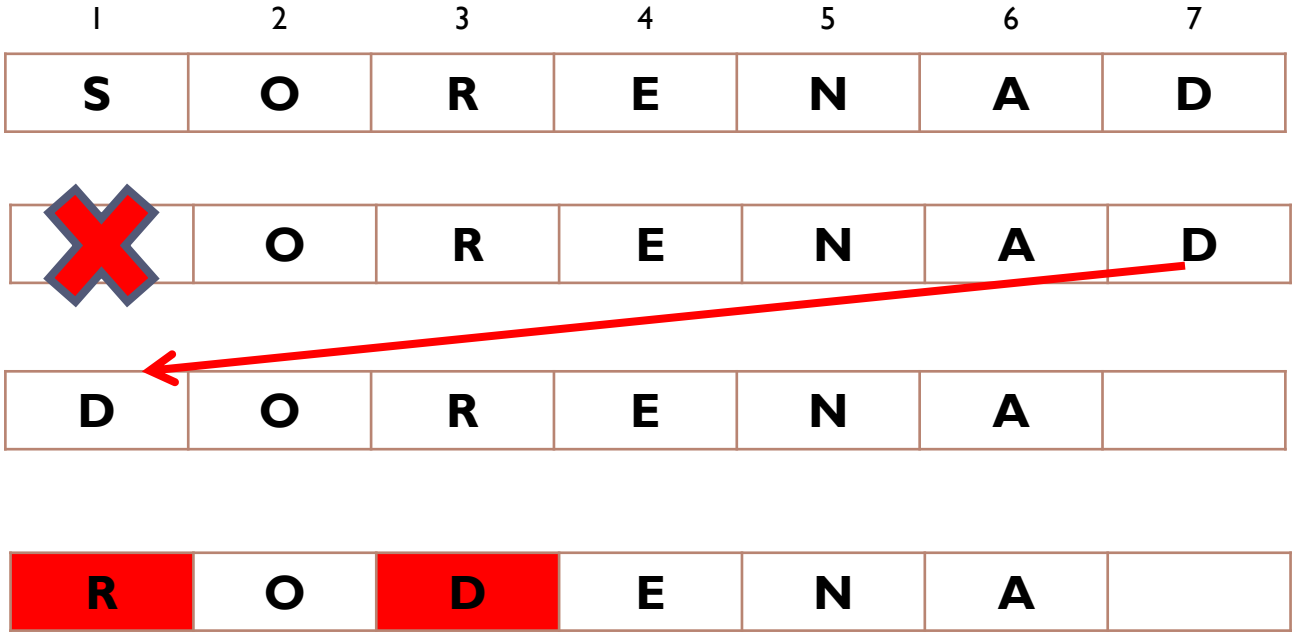


Heap – Refaz a condição de heap

```
void  RefazBaixoCima(Item A[], int k) {

  /* se pai for menor que filho, troca */
  while (k > 1 && A[k/2] < A[k]) {

    Troca(A[k], A[k/2]);

    /* vai para o pai e repete o processo */
    k = k / 2;
  }
```

Heap – Remoção do maior elemento

- Restauração da condição de heap (remoção)
  - Garantir que o valor da chave do pai é maior que dos filhos. Cima para baixo, restaurando o heap



Heap – Refaz a condição de heap

```
void  RefazCimaBaixo(Item A[], int k, int Dir) {
int j;

  while (2*k <= Dir) {
    j = 2*k;
    /* encontra maior filho */
    if (j < Dir && A[j] < A[j+1])
      j++;

    /* testa se pai é maior que filho */
    if (A[k] >= A[j])
      break;

    /* pai é menor que filho, troca posições */
    Troca(A[k], A[j]);
    k = j;
  }
}
```

Heap – Construção do heap

```
void Constroi(Item A[], int N) {
int Esq;

  /* inicia na metade do vetor */
  Esq = N / 2 + 1;

  while (Esq > 1) {
    Esq--;
    RefazCimaBaixo(A, Esq, N);
}
```

Heapsort

```
void Heapsort(Item A[], int n) {
int Esq, Dir;
Item x;

  Constroi(A, n);    /* constroi o heap */

  Esq = 1; Dir = n; /* assumindo elementos em A[1,...,n]*/

  /* ordena o vetor */
  while (Dir > 1) {

    Troca(A[1], A[Dir]);
    Dir--;
    RefazCimaBaixo(A, Esq, Dir);
  }
}
```

Heapsort – Análise

- O procedimento RefazCimaBaixo gasta cerca de log n operações no pior caso.

- Logo, o heapsort gasta um tempo proporcional a O(n log n), no pior caso.

Heapsort

- Vantagens
  - O(n log n).
- Desvantagens
  - Não é estável.