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, paginação de memória, ordenação, simulação de eventos

Fila de Prioridades

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

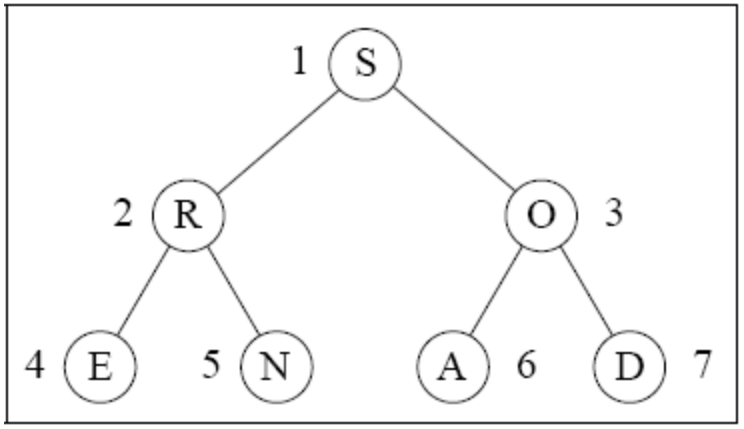
Fila de Prioridades

- Representações
  - Lista encadeada ordenada
  - Lista encadeada não ordenada
  - *Heap*

	Constrói	Insere	Retira máximo	Altera prioridade
Lista ordenada	O(N log N)	O(N)	O(1)	O(N)
Lista não ordenada	O(N)	O(1)	O(N)	O(1)
Heaps	O(N)	O(log N)	O(log N)	O(log N)

Fila de prioridades – representação

- Representação por árvore binária
  - Pai maior que os filhos (a ordem entre os filhos não é especificada)
- Representação vetorial A[1], A[2], …, A[n]
  - Pai na posição *i*, filhos nas posições *2i* e *2i+1*
    - A[i] ≥ A[2i] e A[i] ≥ A[2i+1] para todo i

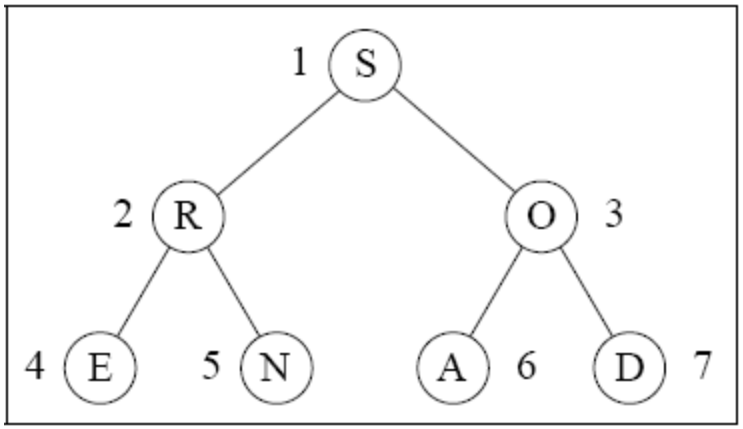


1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*

Fila de prioridades – representação

- Correspondência entre representação em árvore e representação em vetor
  - 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*



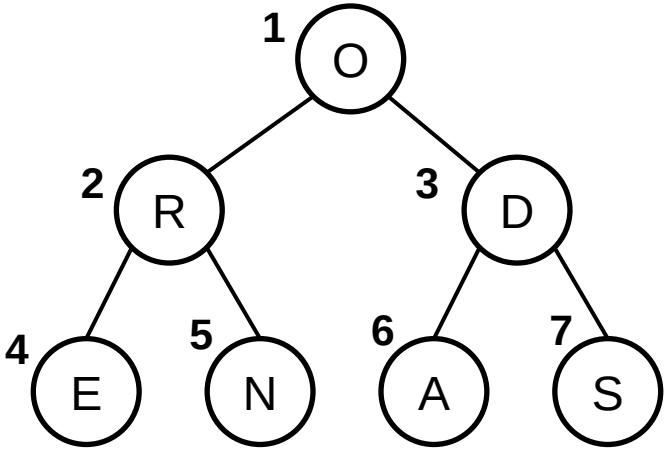
Fila de prioridades – representação

- 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

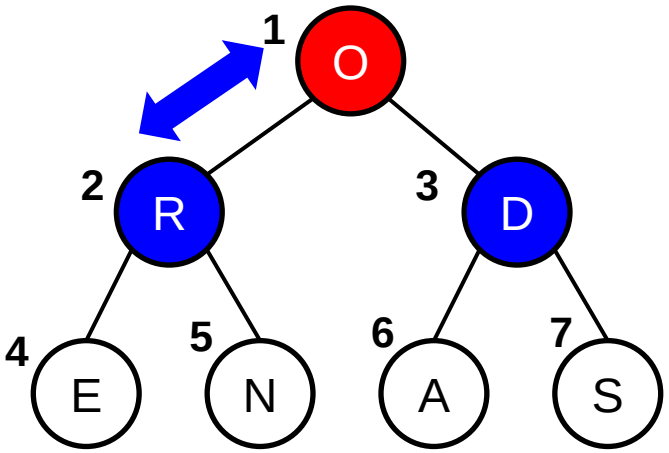
Heap – manutenção da invariante

- Precisamos garantir que o valor da chave do pai é maior que dos filhos
- Se tiver filho maior do que o pai, troca o maior filho com o pai



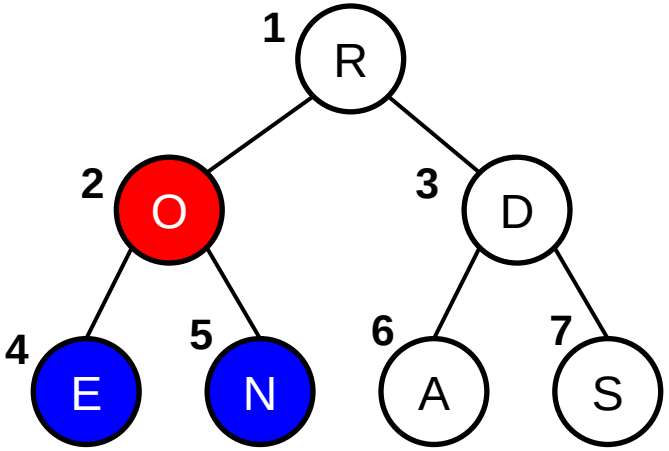
Heap – manutenção da invariante

- Precisamos garantir que o valor da chave do pai é maior que dos filhos
- Se tiver filho maior do que o pai, troca o maior filho com o pai



Heap – manutenção da invariante

- Precisamos garantir que o valor da chave do pai é maior que dos filhos
- Se tiver filho maior do que o pai, troca o maior filho com o pai



Heap – manutenção da invariante

- Precisamos garantir que o valor da chave do pai é maior que dos filhos
- Testa se os elementos A[2i] e A[2i+1] são menores ou igual a A[i]
  - Troca com o maior filho caso contrário

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

Heap – manutenção da invariante

```
void refaz_cima_baixo(struct item *v, int k, int N) {
    int i;
    while(2*k <= N) {
        j = 2*k;
        /* encontra maior filho */
        if(j < N && v[j].chave < v[j+1].chave) { j++; }

        /* testa se pai é maior que filho */
        if(v[k].chave >= v[j].chave) { break; }

        /* troca pai e maior filho, repete */
        troca(v[k], v[j]);
        k = j;
    }
}
```

Heap – Inserindo um elemento

- Usamos **`refaz_cima_baixo`** quando mudamos a chave de um elemento no começo do vetor
- Quando inserimos um elemento no final, temos de trocá-lo com seu pai até que a invariante seja satisfeita

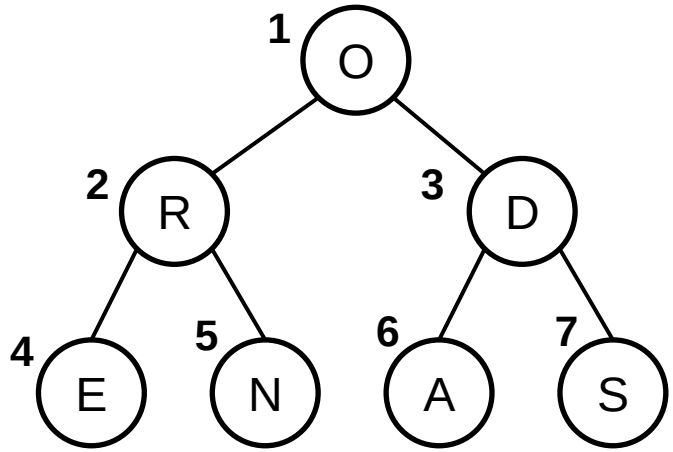
1	2	3	4	5	6	7
O	R	D	E	N	A	
R	O	D	E	N	A	S
R	O	S	E	N	A	D
S	O	R	E	N	A	D

Heap – Inserindo um elemento

```
void refaz_baixo_cima(struct item *v, int k) {
   /* se pai for menor que filho, troca */
   while(k > 1 && v[k/2] < v[k]) {
      troca(v[k], v[k/2]);
      /* vai para o pai e repete o processo */
      k = k/2;
   }
}
```

Transformando um vetor em um *heap*

```
void heap_constroi(struct item *v, int n) {
    k = n/2;
    while(k >= 1) {
        refaz_cima_baixo(v, k, n);
        k--;
    }
}
```



- Número de comparações:

$$\text{seja} m = \lg n$$

$$T(n) = \sum_{1 \le k < m} k 2^{m-k-1} = 2^m - m - 1 < N$$

Fila de Prioridade – Operações

```
void heap_insere(struct item *v, int *n, struct item novo) {
   *n += 1;
   v[*n] = novo;
   refaz_baixo_cima(v, *n);
}

struct item heap_remove_maximo(struct item *v, int *n) {
   troca(v[1], v[*n]);
   *n -= 1;
   refaz_cima_baixo(v, 1, *n);
   return v[*n + 1];
}
```

Heapsort

```
void heapsort(struct item *v, int n) {
    heap_constroi(v, n);
    int k = n;              /* elementos de v[1] a v[n] */
    while(k >= 1) {
        troca(v[k], v[1]);
        k--;
        refaz_cima_baixo(v, 1, k);
    }
}
```

Heapsort – Análise

- O procedimento refaz_cima_baixo gasta no máximo $\lg n$ operações no pior caso

- Logo, o heapsort gasta um tempo proporcional a $O(n \lg(n))$, no pior caso

Vantagens e desvantagens

- Vantagens
  - Comportamento $O(n \lg n)$ no pior caso

- Desvantagens
  - Não é estável
  - Não é tão rápido quanto o quicksort por que o laço interno (`refaz_cima_baixo`) realiza mais operações que o particionamento do quicksort

Exercícios

1. Por que não usar o algoritmo de ordenação por seleção para identificar o k-ésimo menor elemento do vetor?
2. Mesmo com o uso da estratégia da mediana, mostre um vetor de entrada que cai no pior caso do quicksort
3. Um vetor com elementos em ordem decrescente é um heap?
4. Mostre que o heapsort não é estável