COM112 - ALGORITMO E ESTRUTURA DE DADOS II[1]

Pedro Henrique Del Bianco Hokama
UNIFEI

[1] Baseado nos slides elaborados por Charles Ornelas Almeida, Israel Guerra e Nivio Ziviani

Ordenação

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.
- Notação utilizada nos algoritmos:
  - Os algoritmos trabalham sobre os registros de um arquivo ou dados na memória.
  - Cada registro possui uma chave utilizada para controlar a ordenação.
  - Podem existir outros componentes em um registro.

Ordenação

- Estrutura de um registro:

```
typedef long TipoChave;
typedef struct TipoItem{
  TipoChave Chave;
  /* Outros componentes */
} TipoItem;
```

- Qualquer tipo de chave sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado.
- Um método de ordenação é estável se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação.

Ordenação

- Alguns dos métodos de ordenação mais eficientes não são estáveis.
- A estabilidade pode ser forçada quando o método é não-estável.
- Sedgewick (1988) sugere agregar um pequeno índice a cada chave antes de ordenar, ou então aumentar a chave de alguma outra forma

Ordenação

- Classificação dos métodos de ordenação:
  - Interna: arquivo a ser ordenado cabe todo na memória principal.
  - Externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:
  - Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
  - Em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos.
- A maioria dos métodos de ordenação é baseada em comparações das chaves.
- Existem métodos de ordenação que utilizam o princípio da distribuição.

Ordenação por distribuição

- Exemplo de ordenação por distribuição: considere o problema de ordenar um baralho com 52 cartas na ordem:

$$\clubsuit < \diamondsuit < \heartsuit < \spadesuit$$

desempatando por

$$A < 2 < 3 < \ldots < 10 < J < Q < K$$

- Algoritmo:
  1. Distribuir as cartas em treze montes: ases, dois, três, . . ., reis.
  2. Colete os montes na ordem contraria, de forma que o K fique em cima.
  3. Distribua novamente as cartas em quatro montes: paus, ouros, copas e espadas.
  4. Colete os montes na ordem especificada.

Ordenação por distribuição

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**.
- O método não utiliza comparação entre chaves.
- Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte.
- Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode tornar-se proibitiva.
- Sabendo a priori a distribuição das cartas o custo para ordenar um arquivo com n elementos é da ordem de $O(n)$.

Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo n o número registros no arquivo, as medidas de complexidade relevantes são:
  - Número de comparações $C(n)$ entre chaves.
  - Número de movimentações $M(n)$ de itens do arquivo.
- O uso econômico da memória disponível é um requisito primordial na ordenação interna.
- Métodos de ordenação **in situ** são os preferidos.
- Métodos que utilizam listas encadeadas não são muito utilizados.
- Métodos que fazem cópias dos itens a serem ordenados possuem menor importância.

Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - Adequados para pequenos arquivos.
  - Requerem $O(n^2)$ comparações.
  - Produzem programas pequenos.
- Métodos eficientes:
  - Adequados para arquivos maiores.
  - Requerem $O(nlogn)$ comparações.
  - Usam menos comparações.
  - As comparações são mais complexas nos detalhes.
  - Métodos simples são mais eficientes para pequenos arquivos.

Ordenação Interna

- Tipos de dados e variáveis utilizados nos algoritmos de ordenação interna:

```
typedef int TipoIndice;
typedef TipoItem TipoVetor[MAXTAM + 1];
/* MAXTAM + 1 por causa da sentinela */
TipoVetor A;
```

- O índice do vetor vai de 0 até *MaxTam*, devido às chaves sentinelas.
- O vetor a ser ordenado contém chaves nas posições de 1 até n.

Ordenação por Seleção

- Um dos algoritmos mais simples de ordenação.
- Algoritmo:
  - Selecione o menor item do vetor.
  - Troque-o com o item da primeira posição do vetor.
  - Repita essas duas operações com os $n-1$ itens restantes, depois com os $n-2$ itens, até que reste apenas um elemento.

Ordenação por Seleção

- O método é ilustrado abaixo:

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

- As chaves em negrito sofreram uma troca entre si.
- Vamos ordenar o array das posições 1 até n. A posição 0 não é usada na ordenação por seleção.

Ordenação por Seleção

```
1  void Selecao(TipoItem *A, TipoIndice n){
2    TipoIndice i, j, Min;
3    TipoItem x;
4    for (i = 1; i <= n - 1; i++){
5      Min = i;
6      for (j = i + 1; j <= n; j++){
7        if (A[j].Chave < A[Min].Chave){
8          Min = j;
9        }
10     }
11     x = A[Min];
12     A[Min] = A[i];
13     A[i] = x;
14   }
15 }
```


Ordenação por Seleção

- $C(n) = \frac{n^2}{2} - \frac{n}{2}$
- $M(n) = 3(n-1)$
- A atribuição $Min = j$ da linha 8 é executada em média $n \log n$ vezes, Knuth (1973)
- A complexidade do algoritmo de ordenação por seleção é portanto $O(n^2)$

Ordenação por Seleção

Vantagens:

- Custo linar para o número de movimentos de registros.
- É o algoritmo a ser utilizado para arquivos com registros muito grandes.
- É muito interessante para arquivos pequenos.

Desvantagens:

- O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.
- O algoritmo não é **estável**.

Ordenação por Inserção

- Método preferido dos jogadores de cartas.
- Algoritmo:
  - Em cada passo a partir de i=2 faça:
    - Selecione o i-ésimo item da sequência fonte.
    - Coloque-o no lugar apropriado na sequência destino de acordo com o critério de ordenação.

Ordenação por Inserção

- O método é ilustrado abaixo:

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

- As chaves em negrito representam a sequência destino.

Ordenação por Inserção

```
1  void Insercao(TipoItem *A, TipoIndice n){
2    TipoIndice i, j;
3    TipoItem x;
4    for ( i = 2; i <= n ; i++){
5      x = A[i];
6      j = i - 1;
7      A[0] = x; /* sentinela */
8      while ( x.Chave < A[j].Chave){
9        A[j+1] = A[j];
10       j--;
11     }
12     A[j+1] = x;
13   }
14 }
```


Ordenação por Inserção

Considerações sobre o algoritmo:

- O processo de ordenação pode ser terminado pelas condições:
  - Um item com chave menor que o item em consideração é encontrado.
  - O final da sequência destino é atingido à esquerda.
- Solução:
  - Utilizar um registro sentinela na posição zero do vetor.

Ordenação por Inserção - Complexidade

- Seja $C(n)$ a função que conta o número de comparações.
- No laço mais interno, na i-ésima iteração, o valor de C_i é:
  - Melhor caso: $C_i(n) = 1$
  - Pior caso: $C_i(n) = i$
  - Caso médio: $C_i(n) = \frac{1}{i}(1 + 2 + \ldots + i) = \frac{i+1}{2}$
- Assumindo que todas as permutações de n são igualmente prováveis no caso médio, temos:
  - $C(n) = (1 + 1 + \ldots + 1) = n - 1$
  - Pior caso: $C(n) = (2 + 3 + \ldots + n) = \frac{n^2}{2} + \frac{n}{2} - 1$
  - Caso médio: $\frac{1}{2}(3 + 4 + \ldots + n + 1) = \frac{n^2}{4} + \frac{3n}{4} - 1$

Ordenação por Inserção - Complexidade

- Seja $M(n)$ a função que conta o número de movimentações de registros.
- O número de movimentações na i-ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:
  - Melhor caso: $M(n) = (3 + 3 + \ldots + 3) = 3(n-1)$
  - Pior caso: $M(n) = (4 + 5 + \ldots + n + 2) = \frac{n^2}{2} + \frac{5n}{2} - 3$
  - Caso médio: $M(n) = \frac{1}{2}(5 + 6 + \ldots + n + 3) = \frac{n^2}{4} + \frac{11n}{4} - 3$

Ordenação por Inserção - Considerações

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o arquivo está "quase" ordenado.
- É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear.
- O algoritmo de ordenação por inserção é estável.

Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A idéia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- Os resultados são combinados para produzir a solução final.

Quicksort

- A parte mais delicada do método é o processo de partição.
- O vetor $A[Esq \ldots Dir]$ é rearranjado por meio da escolha arbitrária de um **pivô** x.
- O vetor A é particionado em duas partes:
  - A parte esquerda com chaves menores ou iguais a x.
  - A parte direita com chaves maiores ou iguais a x.

Quicksort

- Algoritmo para o particionamento:
  1. Escolha arbitrariamente um **pivô** x.
  2. Percorra o vetor a partir da esquerda até que $A[i] \geq x$.
  3. Percorra o vetor a partir da direita até que $A[j] \leq x$.
  4. Troque $A[i]$ com $A[j]$.
  5. Continue este processo até os apontadores i e j se cruzarem.
- Ao final, o vetor $A[Esq..Dir]$ está particionado de tal forma que:
  - Os itens em $A[Esq], A[Esq+1], \ldots, A[j]$ são menores ou iguais a x.
  - Os itens em $A[i], A[i+1], \ldots, A[Dir]$ são maiores ou iguais a x.

Quicksort

- Ilustração do processo de partição:

1	2	3	4	5	6
O	R	$\mathbf{D}$	E	N	A
A	R	$\mathbf{D}$	E	N	O
A	$\mathbf{D}$	R	E	N	O

- O pivô x é escolhido como sendo $A[(i+j)div2]$.
- Como inicialmente $i = 1$ e $j = 6$, então $x = A[3] = D$.
- Ao final do processo de partição i e j se cruzam em $i = 3$ e $j = 2$.

Partição

```
1  void Particao (TipoIndice Esq, TipoIndice Dir,
2  TipoIndice *i, TipoIndice *j, TipoItem *A){
3    TipoItem x, w;
4    *i = Esq;
5    *j = Dir;
6    x = A[(*i + *j) / 2 ] ; /* obtem o pivo x */
7    do{
8      while(x.Chave > A[*i].Chave) (*i)++;
9      while(x.Chave < A[*j].Chave) (*j)--;
10     if(*i <= *j){
11       w = A[*i]; A[*i] = A[*j]; A[*j] = w;
12       (*i)++; (*j)--;
13     }
14   }while(*i <= *j);
15 }
```


Partição

- O anel interno do procedimento Particao é extremamente simples.
- Razão pela qual o algoritmo Quicksort é tão rápido.

Quicksort

```
1  void Ordena(TipoIndice Esq, TipoIndice Dir,
2              TipoItem *A){
3    TipoIndice i, j;
4    Particao (Esq, Dir, &i, &j, A);
5    if(Esq < j) Ordena(Esq, j, A);
6    if(i < Dir) Ordena(i, Dir, A);
7  }
8
9  void QuickSort(TipoItem *A, TipoIndice n){
10   Ordena(1, n, A);
11 }
```


Quicksort: Análise

- Seja C(n) a função que conta o número de comparações.
- Pior caso:

$$C(n) = O(n^2)$$

- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
- Isto faz com que o procedimento Ordena seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada.
- O pior caso pode ser evitado empregando pequenas modificações no algoritmo.
- Para isso basta escolher três itens quaisquer do vetor e usar a **mediana dos três** como pivô.

Quicksort: Análise

- Melhor caso:

$$C(n) = 2C(n/2) + n = n \log n - n + 1$$

- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
- Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):

$$C(n) \approx 1,386 n \log n - 0,846 n$$

• Isso significa que em média o tempo de execução do Quicksort é O(n log n).

Quicksort

- Vantagens:
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - Requer cerca de $n \log n$ comparações em média para ordenar n itens.
- Desvantagens:
  - Tem um pior caso $O(n^2)$ comparações.
  - Sua implementação é muito delicada e difícil:
    - ⋆ Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
  - O método não é estável.

Heapsort

- Possui o mesmo princípio de funcionamento da ordenação por seleção.
- Algoritmo:
  1. Selecione o menor item do vetor.
  2. Troque-o com o item da primeira posição do vetor.
  3. Repita estas operações com os $n-1$ itens restantes, depois com os $n-2$ itens, e assim sucessivamente.
- O custo para encontrar o menor (ou o maior) item entre n itens é $n-1$ comparações. (em uma busca linear)
- Isso pode ser reduzido utilizando uma fila de prioridades.

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 - TAD

- 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. Une duas filas de prioridades em uma única.

Filas de Prioridades - Representação

- Representação através de uma lista linear ordenada:
  - Neste caso, Constrói leva tempo $O(n \log n)$.
  - Insere é $O(n)$.
  - Retira é $O(1)$.
  - Unir é $O(n)$.
- Representação é através de uma lista linear não ordenada:
  - Neste caso, Constrói tem custo linear.
  - Insere é $O(1)$.
  - Retira é $O(n)$.
  - Unir é $O(1)$ para apontadores e $O(n)$ para arranjos.

Filas de Prioridades - 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 Unir 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 - Algoritmos de Ordenação

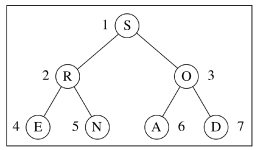
- As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
- Basta utilizar repetidamente a operação Insere para construir a fila de prioridades.
- Em seguida, utilizar repetidamente a operação Retira para receber os itens na ordem reversa.
- O uso de listas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

Heap

- É uma seqüência de itens com chaves $c[1], c[2], \ldots, c[n]$, tal que:

$$c[i] \geq c[2i], \quad (1)$$
$$c[i] \geq c[2i+1], \quad (2)$$

  para todo $i = 1, 2, \ldots, n/2$.
- A definição pode ser facilmente visualizada em uma árvore binária completa:



Heap

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

Heap

- 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 array:

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

Heap

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

- 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 $\lfloor i/2 \rfloor$.

Heap

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

Heap

	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

- Os itens de $A[4]$ a $A[7]$ estão trivialmente atendendo as condições.
- O heap é estendido para a esquerda ($Esq = 3$), englobando o item $A[3]$, pai dos itens $A[6]$ e $A[7]$

Heap

- A condição de heap é violada:
  - O heap é refeito trocando os itens D e S.
- O item R é incluindo 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 violada:
  - O heap é refeito trocando os itens O e S, encerrando o processo.

Heap

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

```
1  TipoItem Max(TipoItem *A){
2    return (A[1]);
3  }
```


Heap

Função para refazer o heap:

```
1  void Refaz(TipoIndice Esq, TipoIndice Dir,
2             TipoItem *A){
3    TipoIndice i = Esq;
4    int j; TipoItem x;
5    j = i * 2;
6    x = A[i];
7    while (j <= Dir){
8      if (j < Dir)
9        if (A[j].Chave < A[j+1].Chave) j++;
10     if (x.Chave >= A[j].Chave) break;
11     A[i] = A[j]; i = j; j = i*2;
12   }
13   A[i] = x;
14 }
```


Heap

Função para construir o heap:

```
1  void Constroi(TipoItem *A, TipoIndice n){
2    TipoIndice Esq;
3    Esq = n / 2 + 1;
4    while (Esq > 1){
5      Esq--;
6      Refaz(Esq, n, A);
7    }
8  }
```

Heap

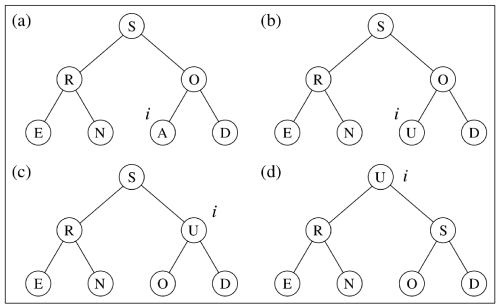
Função que implementa a operação de retirar o item com maior chave:

```
1  TipoItem RetiraMax(TipoItem *A,
2                     TipoIndice *n){
3    TipoItem Maximo;
4    if (*n < 1)
5      printf("Erro: heap vazio \n" );
6    else{
7      Maximo = A[1];
8      A[1] = A[*n];
9      (*n)--;
10     Refaz(1, *n, A);
11   }
12   return Maximo;
13 }
```


Heap

```
1  void AumentaChave(TipoIndice i,
2           TipoChave ChaveNova, TipoItem *A){
3    TipoItem x;
4    if(ChaveNova < A[i].Chave){
5      printf("ChaveNova menor que atual \n");
6      return;
7    }
8    A[i].Chave = ChaveNova;
9    while(i > 1 && A[i/2].Chave < A[i].Chave){
10     x = A[i/2];
11     A[i/2] = A[i];
12     A[i] = x;
13     i = i/2;
14   }
15 }
```


- 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 $O(\log n)$

Heap

Função que implementa a operação de inserir um novo item no heap:

```
1  void Insere (TipoItem *x , TipoItem *A,
2               TipoIndice *n){
3    (*n)++;
4    A[*n] = *x;
5    A[*n].Chave = INT_MIN;
6    AumentaChave(*n, x->Chave, A);
7  }
```

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 $A[1], A[2], \ldots, A[n-1]$.
  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*					

- 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 - Implementação

```
1  void Heapsort(TipoItem *A, TipoIndice n){
2    TipoIndice Esq, Dir;
3    TipoItem x;
4    Constroi(A, n);   /* constroi o heap */
5    Esq = 1;
6    Dir = n;
7    while(Dir > 1){
8      /* ordena o vetor */
9      x = A[1];
10     A[1] = A[Dir];
11     A[Dir] = x;
12     Dir--;
13     Refaz(Esq, Dir, A);
14   }
15 }
```


Heapsort - Análise

- O procedimento Refaz gasta cerca de $\log n$ operações, no pior caso.
- Logo, Heapsort gasta um tempo de execução proporcional a $n \log n$, no pior caso.

Heapsort

- Vantagens:
  - O comportamento do Heapsort é sempre $O(n \log n)$, qualquer que seja a entrada.
- Desvantagens:
  - O laço 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.

Comparação entre os Métodos

Complexidade:

	Melhor Caso	Caso Médio	Pior Caso
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
BubleSort	$O(n)$	$O(n^2)$	$O(n^2)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

MergeSort

baseado nos slides do Prof. Rafael Schouery

Pesquisa em Memória Primária

- Introdução - Conceitos Básicos
- Pesquisa Sequencial
- Pesquisa Binária
- Árvores de Pesquisa

Introdução - Conceitos Básicos

- Estudo de como recuperar informação a partir de uma grande massa de informação previamente armazenada.
- A informação é dividida em **registros**.
- Cada registro possui uma chave para ser usada na pesquisa.
- Objetivo da pesquisa:
  Encontrar uma ou mais ocorrências de registros com chaves iguais à chave de pesquisa.
- **Pesquisa com sucesso** X **Pesquisa sem sucesso**.

Introdução - Conceitos Básicos

- Conjunto de registros ou arquivos: tabelas
- **Tabela**:
  associada a entidades de vida curta, criadas na memória interna durante a execução de um programa.
- **Arquivo**:
  geralmente associado a entidades de vida mais longa, armazenadas em memória externa.
- **Distinção não é rígida**:
  tabela: arquivo de índices
  arquivo: tabela de valores de funções.

Escolha do Método de Pesquisa mais Adequado a uma Determinada Aplicação

- Depende principalmente
  1. Quantidade dos dados envolvidos.
  2. Arquivo estar sujeito a inserções e retiradas frequentes.

Se conteúdo do arquivo é estável é importante minimizar o tempo de pesquisa, sem preocupação com o tempo necessário para estruturar o arquivo

Algoritmos de Pesquisa: Tipos Abstratos de Dados

- É importante considerar os algoritmos de pesquisa como **tipos abstratos de dados**, com um conjunto de operações associado a uma estrutura de dados, de tal forma que haja uma independência de implementação para as operações.
- Operações mais comuns:
  1. Inicializar a estrutura de dados.
  2. Pesquisar um ou mais registros com determinada chave.
  3. Inserir um novo registro.
  4. Retirar um registro específico.
  5. Ordenar um arquivo para obter todos os registros em ordem de acordo com a chave.
  6. Ajuntar dois arquivos para formar um arquivo maior.

Dicionário

- Nome comumente utilizado para descrever uma estrutura de dados para pesquisa.
- **Dicionário** é um tipo abstrato de dados com pelo menos as operações:
  1. Inicializa
  2. Pesquisa
  3. Insere
  4. Retira
- Analogia com um dicionário da língua portuguesa:
  1. Chaves = Palavras
  2. Registros = entradas associadas com cada palavra
     - pronúncia
     - definição
     - sinônimos
     - outras informações

Pesquisa Sequencial

- **Método de pesquisa mais simples**: a partir do primeiro registro, pesquise sequencialmente até encontrar a chave procurada; então pare.

- Armazenamento de um conjunto de registros por meio do tipo estruturado arranjo:

```
1  #define MAXN 10
2
3  typedef long TipoChave;
4
5  typedef struct TipoRegistro{
6    TipoChave Chave;
7    /* outros componentes */
8  } TipoRegistro;
9
10 typedef int TipoIndice;
11
12 typedef struct TipoTabela{
13   TipoRegistro Item [MAXN + 1];
14   TipoIndice n;
15 } TipoTabela;
```


Pesquisa Sequencial

```
1  void Inicializa (TipoTabela *T){
2  ??
3  }
4
5  TipoIndice Pesquisa(TipoChave x,
6                      TipoTabela *T){
7  ??
8  }
9
10 void Insere (TipoRegistro Reg,
11              TipoTabela *T){
12 ??
13 }
```

Pesquisa Sequencial

- Pesquisa retorna o índice do registro que contém a chave x;
- Caso não esteja presente, o valor retornado é zero.
- A implementação não suporta mais de um registro com uma mesma chave.
- Para aplicações com esta característica é necessário incluir um argumento a mais na função Pesquisa para conter o índice a partir do qual se quer pesquisar.

Pesquisa Sequencial

- Utilização de um registro sentinela na posição zero do array:
  1. Garante que a pesquisa sempre termina:
     se o índice retornado por Pesquisa for zero, a pesquisa foi sem sucesso.
  2. Não é necessário testar se i > 0, devido a isto:
     - o anel interno da função Pesquisa é extremamente simples: o índice i é decrementado e a chave de pesquisa é comparada com a chave que está no registro.
     - isto faz com que esta técnica seja conhecida como **pesquisa sequencial rápida**.

Pesquisa Sequencial: Análise

- Pesquisa com sucesso:

$$\text{melhor caso: } C(n) = 1$$
$$\text{pior caso: } C(n) = n$$
$$\text{caso médio: } C(n) = (n+1)/2$$

- Pesquisa sem sucesso:

$$C'(n) = n + 1.$$

- O algoritmo de pesquisa sequencial é a melhor escolha para o problema de pesquisa em tabelas com até **25 registros**.

Pesquisa Binária

- Pesquisa em tabela pode ser mais eficiente se registros forem mantidos em ordem
- Para saber se uma chave está presente na tabela
  1. Compare a chave com o registro que está na posição do meio da tabela.
  2. Se a chave é menor então o registro procurado está na primeira metade da tabela
  3. Se a chave é maior então o registro procurado está na segunda metade da tabela.
  4. Repita o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é diferente da procurada, significando uma pesquisa sem sucesso.

```
1  TipoIndice Binaria (TipoChave x,
2                      TipoTabela *T){
3    TipoIndice i, Esq, Dir;
4    if(T->n == 0) return 0;
5    else{
6      Esq = 1;
7      Dir = T->n;
8      do{
9        i = (Esq + Dir) / 2;
10       if(x > T->Item[i].Chave) Esq = i + 1;
11       else Dir = i - 1;
12     }while(x != T->Item[i].Chave &&
13            Esq <= Dir);
14     if(x == T->Item[i].Chave) return i;
15     else return 0;
16   }
17 }
```


Pesquisa Binária: Análise

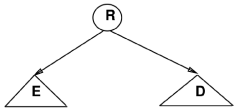
- A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio.
- Logo: o número de vezes que o tamanho da tabela é dividido ao meio é cerca de $\log n$.
- **Ressalva:** o custo para manter a tabela ordenada é alto: a cada inserção na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes.
- Consequentemente, a pesquisa binária não deve ser usada em aplicações muito dinâmicas.

Árvores de Pesquisa

- A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação.
- Particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:
  1. Acesso direto e sequencial eficientes.
  2. Facilidade de inserção e retirada de registros.
  3. Boa taxa de utilização de memória.
  4. Utilização de memória primária e secundária.

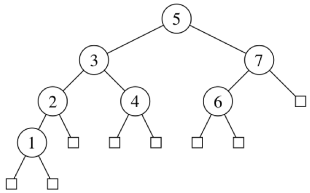
Árvores Binárias de Pesquisa sem Balanceamento

- Para qualquer nó que contenha um registro



- Temos a relação invariante
  1. Todos os registros com chaves menores estão na subárvore à esquerda.
  2. Todos os registros com chaves maiores estão na subárvore à direita.

Árvores Binárias de Pesquisa sem Balanceamento



- O **nível** do nó raiz é 0.
- Se um nó está no nível i então a raiz de suas subárvores estão no nível $i + 1$.
- A **altura** de um nó é o comprimento do caminho mais longo deste nó até um nó folha.
- A altura de uma árvore é a altura do nó raiz.

Implementação

```
1  typedef long TipoChave;
2  typedef struct TipoRegistro{
3    TipoChave Chave;
4    /* outros componentes */
5  } TipoRegistro;
6
7  typedef struct TipoNo *TipoApontador;
8
9  typedef struct TipoNo{
10   TipoRegistro Reg;
11   TipoApontador Esq, Dir;
12 } TipoNo;
```


Procedimento para Pesquisar na Árvore Uma Chave x

1. Compare-a com a chave que está na raiz.
2. Se x é menor, vá para a subárvore esquerda.
3. Se x é maior, vá para a subárvore direita.
4. Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha é atingido.
5. Se a pesquisa tiver sucesso retorna o registro, senão devolve NULL.

Procedimento para Pesquisar na Árvore Uma Chave x

```
1  TipoRegistro * Pesquisa(TipoChave x,
2                          TipoApontador *p){
3    if(*p == NULL){return NULL;}
4    if(x < (*p)->Reg.Chave)
5      return Pesquisa(x, &(*p)->Esq);
6    if(x > (*p)->Reg.Chave)
7      return Pesquisa(x, &(*p)->Dir);
8    else return &((*p)->Reg);
9  }
```

Procedimento para Inserir na Árvore

- Atingir um apontador nulo em um processo de pesquisa significa uma pesquisa sem sucesso.
- O apontador nulo atingido é o ponto de inserção.

Procedimento para Inserir na Árvore

```
1  void Insere(TipoRegistro x, TipoApontador *p){
2    if(*p == NULL){
3      *p = (TipoApontador) malloc(sizeof(TipoNo));
4      (*p)->Reg = x;
5      (*p)->Esq = NULL;
6      (*p)->Dir = NULL;
7      return;
8    }
9    if(x.Chave < (*p)->Reg.Chave){
10     Insere(x, &(*p)->Esq);
11     return;
12   }
13   if(x.Chave > (*p)->Reg.Chave)
14     Insere (x, &(*p)->Dir);
15   else
16     printf("Erro : Registro ja existe na arvore \n");
17 }
```


Procedimentos para Inicializar e Criar a Árvore

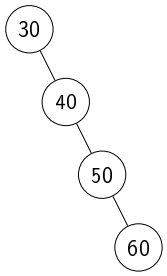
```
1  void Inicializa (TipoApontador *p){
2    *p = NULL;
3  }
4
5  int main(int argc, char *argv[]){
6    TipoApontador Dicionario;
7    TipoRegistro x;
8    Inicializa(&Dicionario);
9    scanf( "%ld%*[^\n]", &x.Chave);
10   while(x.Chave > 0){
11     Insere (x,&Dicionario ) ;
12     scanf("%ld%*[^\n]" , &x.Chave) ;
13   }
14 }
```


Exercícios

- Imprimir todos os registros de uma árvore em ordem de chaves
- Remover um nó

Árvore Binária de Busca

- Uma sequência de inserções pode gerar uma árvore muito desbalanceada.



No pior caso:

- Inserção: $O(n)$
- Busca: $O(n)$

Solução?
Manter a árvore suficientemente balanceada.

Árvores AVL

- Nomeada em homenagem aos seus inventores soviéticos Georgy **A**delson-**V**elsky e Evgenii **L**andis
- A altura de um nó h é o comprimento do caminho mais longo até uma folha, e denotado por $H(n)$
- A altura de uma árvore vazia é -1
- O fator de Balanceamento de um nó n é denominado por:

$$B(n) = H(n.dir) - H(n.esq)$$

- Uma árvore é dita **AVL** se para todo nó n na árvore,

$$B(n) \in \{-1, 0, 1\}$$

- A altura h de uma árvore *AVL* com n é tal que [2]:

$$\lg(n+1) \leq h < 1.44 \lg(n+2) - 0.328$$

- As operações de busca são idênticas as de uma árvore binária de busca. Qual a complexidade?
- As inserções e remoções precisam ser feitas de forma a preservar a propriedade de árvore AVL
- Essa propriedade é mantida através de rotações na árvore

[2] Knuth, Donald E. (2000). Sorting and searching (2. ed)

Rotações

Exercício em aula!

Árvores Rubro Negras

Slides do Prof. Rafael do IC - UNICAMP

Árvores B

Slides do Prof. Rafael do IC - UNICAMP

Árvores B

Remoção

A remoção é mais complicada que a inserção

- Ela pode ocorrer em qualquer lugar da árvore
- Cada nó precisa continuar com pelo menos $t - 1$ chaves
  - exceto a raiz que tem que ter pelo menos 1 chave

O algoritmo tenta resolver esse problema garantindo que os nós no caminho da remoção tem pelo menos t chaves

- nesse caso não há problema em remover
- nem sempre consegue, mas existe uma solução
- eventualmente junta dois nós vizinhos com $t - 1$ chaves
  - formando um nó com $2t - 1$ chaves

Variantes

Árvores B^*: Cada nó tem interno, exceto a raiz, estão pelo menos 2/3 (ao invés de 1/2)
Árvores B^+: Mantém os registros apenas nas folhas, e cópias das chaves nos nós internos.

Árvore B^+

- Semelhante à pesquisa em árvore B,
- A pesquisa sempre leva a uma página folha,
- A pesquisa não para se a chave procurada for encontrada em uma página índice. O apontador da direita é seguido até que se encontre uma página folha.

Árvore B^+

Inserção na árvore B^+

- Semelhante à inserção na árvore B,
- Diferença: quando uma folha é dividida em duas, o algoritmo promove uma cópia da chave que pertence ao registro do meio para a página pai no nível anterior, retendo o registro do meio na página folha da direita.

Remoção na árvore B^+

- Relativamente mais simples que em uma árvore B,
- Todos os registros são folhas,
- Desde que a folha fique com pelo menos metade dos registros, as páginas dos índices não precisam ser modificadas, mesmo se uma cópia da chave que pertence ao registro a ser retirado esteja no índice.

Acesso Concorrente em Árvore B^+

- Acesso simultâneo a banco de dados por mais de um usuário.
- Concorrência aumenta a utilização e melhora o tempo de resposta do sistema.
- O uso de árvores B* nesses sistemas deve permitir o processamento simultâneo de várias solicitações diferentes.
- Necessidade de criar mecanismos chamados protocolos para garantir a integridade tanto dos dados quanto da estrutura.

Acesso Concorrente em Árvore B^+

- Página segura: não há possibilidade de modificações na estrutura da árvore como consequência de inserção ou remoção.
  - inserção: página segura se o número de chaves é igual a $2m$,
  - remoção: página segura se o número de chaves é maior que m.
- Os algoritmos para acesso concorrente fazem uso dessa propriedade para aumentar o nível de concorrência.

Acesso Concorrente em Árvore B^+

- Quando uma página é lida, a operação de recuperação a trava, assim, outros processos, não podem interferir com a página.
- A pesquisa continua em direção ao nível seguinte e a trava é liberada para que outros processos possam ler a página .
- Processo leitor: executa uma operação de recuperação
- Processo modificador: executa uma operação de inserção ou retirada.

Acesso Concorrente em Árvore B^+

- Dois tipos de travamento:
  - Travamento para leitura: permite um ou mais leitores acessarem os dados, mas não permite inserção ou retirada.
  - Travamento exclusivo: nenhum outro processo pode operar na página e permite qualquer tipo de operação na página.

Transformação de Chave (*Hashing*)

- Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.
- *Hash* significa:
  - Fazer picadinho de carne e vegetais para cozinhar.
  - Fazer uma bagunça. (Webster's New World Dictionary)

- Um método de pesquisa com o uso da transformação de chave é constituído de duas etapas principais:
  1. Computar o valor da função de transformação, a qual transforma a chave de pesquisa em um endereço da tabela.
  2. Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método para lidar com colisões.
- Qualquer que seja a função de transformação, algumas colisões irão ocorrer fatalmente, e tais colisões têm de ser resolvidas de alguma forma.
- Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entradas da tabela, existe uma alta probabilidade de haver colisões.

- O **paradoxo do aniversário** (Feller,1968, p. 33), diz que em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que 2 pessoas comemorem aniversário no mesmo dia.
- Assim, se for utilizada uma função de transformação uniforme que enderece 23 chaves randômicas em uma tabela de tamanho 365, a probabilidade de que haja colisões é maior do que 50%.

Funções de Transformação

- Uma função de transformação deve mapear chaves em inteiros dentro do intervalo $[0 \ldots M-1]$, onde M é o tamanho da tabela.
- A função de transformação ideal é aquela que:
  1. Seja simples de ser computada.
  2. Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.

Método mais Usado

- Usa o resto da divisão por M.

$$h(K) = K \mod M$$

  onde K é um inteiro correspondente à chave.
- Cuidado na escolha do valor de M. M deve ser um número primo, mas não qualquer primo: devem ser evitados os números primos obtidos a partir de

$$b^i \pm j$$

  onde b é a base do conjunto de caracteres (geralmente $b = 64$ para BCD, 128 para ASCII, 256 para EBCDIC, ou 100 para alguns códigos decimais), e i e j são pequenos inteiros.

Transformação de Chaves Não Numéricas

- As chaves não numéricas devem ser transformadas em números:

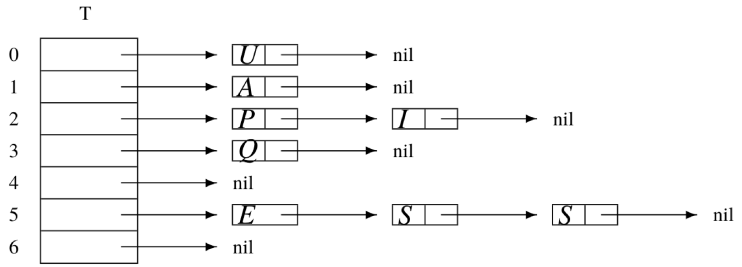
$$K = \sum_{i=1}^{n} Chave[i] \times p[i]$$

- n é o número de caracteres da chave.
- $Chave[i]$ corresponde à representação ASCII do i-ésimo caractere da chave.
- $p[i]$ é um inteiro de um conjunto de pesos gerados randomicamente para $1 \leq i \leq n$.
- Vantagem de usar pesos: Dois conjuntos diferentes de $p_1[i]$ e $p_2[i]$, $1 \leq i \leq n$, leva a duas funções $h_1(K)$ e $h_2(K)$ diferentes.


```
1  void GeraPesos(TipoPesos p){
2    int i;
3    struct timeval semente;
4    /*Utilizar o tempo como semente para a funcao srand()*/
5    gettimeofday(&semente, NULL);
6    srand((int) (semente.tv_sec + 1000000*semente.tv_usec));
7    for (i=0; i<n; i++)
8      p[i] = 1+(int) (10000.0*rand( ) / ( RAND_MAX+1.0));
9  }
10
11 typedef char TipoChave[N];
12 TipoIndice h(TipoChave Chave, TipoPesos p){
13   int i;
14   unsigned int Soma = 0;
15   int comp = strlen (Chave);
16   for (i=0; i < comp; i++)
17     Soma += (unsigned int) Chave[i] * p[i];
18   return (Soma % M) ;
19 }
```


Listas Encadeadas

- Uma das formas de resolver as **colisões** é construir uma lista linear encadeada para cada endereço da tabela. Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.
- Exemplo: Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação $h(Chave) = Chave \mod M$ é utilizada para $M = 7$, o resultado da inserção das chaves *PESQUISA* na tabela é o seguinte:
  $h(A) = h(1) = 1, h(E) = h(5) = 5, h(S) = h(19) = 5$, e assim por diante.



Análise

- Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada de T, então o comprimento esperado de cada lista encadeada é N/M, onde N representa o número de registros na tabela e M o tamanho da tabela.
- Logo: as operações Pesquisa, Insere e Retira custam $O(1 + N/M)$ operações em média, onde a constante 1 representa o tempo para encontrar a entrada na tabela e N/M o tempo para percorrer a lista. Para valores de M próximos de N, o tempo se torna constante, isto é, independente de N.