Aula 8: Busca, Ordenação, Hash

Professor(a): Virgínia Fernandes Mota
http://www.dcc.ufmg.br/∼virginiaferm

ALGORITMOS E ESTRUTURAS DE DADOS - SETOR DE INFORMÁTICA

Busca

Busca

- Vamos discutir diferentes estratégias para efetuar a busca de um elemento em um determinado conjunto de dados.
- Vetores x Árvores Binárias de Busca.

- Dado um vetor *vet* com *n* elementos, desejamos saber de um determinado elemento *elem* está ou não presente no vetor.
- Forma mais simples: Busca linear

```
int busca (int n, int *vet, int elem){

    int i;
    for (i = 0; i < n; i++){
        if (elem == vet[i])
            return i;
    }
    return -1;
}
```

Busca em Vetor

- Busca linear: Muito ineficiente!
- Complexidade $O(n)$: No pior caso, precisaremos realizar n comparações.
- É possível melhorar?

Busca em Vetor

- Assumindo que os elementos do vetor estão ordenados (ordem crescente).

```
int busca_ord(int n, int *vet, int elem){
    int i;

    for (i = 0; i < n; i++){
        if (elem == vet[i])
            return i;
        else if (elem < vet[i])
            return -1;
    }
    return -1;
}
```

- Mas o algoritmo continua sendo linear - $O(n)$.
- No entanto, se os elementos estão ordenados, existe uma busca muito mais eficiente.

Busca Binária

- Com os elementos ordenados, podemos aplicar um algoritmo mais eficiente: Busca Binária.

```c
int busca_bin(int n, int *vet, int elem){
    int ini = 0;
    int fim =  n-1;
    int meio;

    while (ini <= fim){
        meio = (ini + fim) / 2;
        if (elem < vet[meio])
            fim = meio - 1;
        else if (elem > vet[meio])
            ini = meio +1;
        else
            return meio;
    }
    return -1;
}
```

- Qual a complexidade desse algoritmo?

Repetições	Tamanho do problema
1	n
2	n/2
3	n/4
...	...
log n	1

- Assim, são necessárias *logn* repetições.
- Como fazemos um número constante de comparações a cada ciclo (duas), podemos concluir que a ordem de complexidade desse algoritmo $O(logn)$.

Árvore Binária de Busca

- O algoritmo de busca binária tem bom desempenho computacional e deve ser usado sempre que temos dados ordenados armazenados em um vetor.
- Contudo, se precisarmos inserir e remover elementos da estrutura e ao mesmo tempo dar suporte a funções de busca eficientes, a estrutura de vetor não se mostra adequada.
- Precisamos de uma estrutura dinâmica e eficiente: Árvore Binária de Busca - $O(logn)$.

Árvore Binária de Busca

- Uma Árvore Binária de Busca é tal que ou a árvore é dita vazia ou seu nó raiz contém uma chave e:
  - Todas as chaves da sub-árvore esquerda são menores que a chave da raiz.
  - Todas as chaves da sub-árvore direita são maiores que a chave raiz.
  - As sub-árvore direita e esquerda são também Árvore Binária de Busca.

Operações em Árvore Binária de Busca

- Vamos analisar as funções para árvores binária de busca:
  - busca: função que busca um elemento na árvore;
  - insere: função que insere um novo elemento na árvore;
  - retira: função que retira um elemento da árvore.

Operação de Busca

```
Arv *abb_busca(Arv *r, int v){
    if (r == NULL) return NULL;
    else if (r->info > v) return abb_busca(r->esq, v);
    else if (r->info < v) return abb_busca(r->dir, v);
    else return r;
}
```

Operação de Inserção

```
Arv *abb_insere(Arv *a, int v){
    if (a == NULL){
        a = (Arv *) malloc (sizeof(Arv));
        a->info = v;
        a->esq = a->dir = NULL;
    }
    else if (v < a->info)
        a->esq = abb_insere(a->esq, v);
    else
        a->dir = abb_insere(a->dir, v);
    return a;
}
```

Operação de Remoção

- A operação de remoção é mais complexa que a inserção.
- Existem três situações possíveis:
  - Retirar uma raiz que é um nó folha.
  - Retirar uma raiz que tem apenas um filho.
  - Retirar uma raiz que tem dois filhos.

Operação de Remoção

```c
Arv *abb_retira(Arv *r, int v){
    if (r == NULL)
        return NULL;
    else if (r->info > v)
        r->esq = abb_retira(r->esq, v);
    else if (r->info < v)
        r->dir = abb_retira(r->dir, v);
    else{
        //elemento sem filhos
        if (r->esq == NULL && r->dir == NULL)
            free(r);
            r = NULL;
        //elemento tem filho à direita
        else if (r->esq == NULL){
            Arv *t = r;
            r = r->dir;
            free(t);
        }
        //elemento tem filho à esquerda
      else if (r->dir == NULL){
            Arv *t = r;
            r = r->esq;
            free(t);
        }
//...
```

Operação de Remoção

```c
//...
        //elemento tem dois filhos
        else{
            Arv *f = r->esq;
            while (f->dir != NULL){
                f = f->dir;
            }
            r->info = f->info;
            f->info = v;
            r->esq = abb_retira(r->esq, v);
        }
    }
    return r;
}
```

Árvores Binárias de Busca

- É fácil prever que, após várias operações de inserção/remoção, a árvore tende a ficar desbalanceada.
- Para manter o balanceamento da árvore, devemos tentar manter a altura da árvore no mínimo.
- Algoritmos para manter a árvore balanceada são mais complexos, como já vimos!

Ordenação

Ordenação

- Definição do problema: Dada uma coleção de n elementos, representada em um vetor de 0 a n-1, deseja-se obter uma outra coleção, cujos elementos estejam ordenados segundo algum critério de comparação entre os elementos.
- Tipos de algoritmos que serão discutidos: Iterativos e Recursivos
- Alguns exemplos: InsertionSort, **BubbleSort**, RadixSort, HeapSort, **MergeSort**, **QuickSort**.

BubbleSort

- Os elementos do vetor devem ser trocados entre si para que fiquem na ordem desejada.

```
void bubbleSort(int *p, int t){
    int i, j, aux;
    for ( i=0; i < t; i++ ){
        for ( j=t-1; j > i; j-- ){
            if ( p[j] < p[j-1]){
                aux = p[j];
                p[j] = p[j-1];
                p[j-1] = aux;
            }
        }
    }
}
```

http://www.youtube.com/watch?v=lyZQPjUT5B4

MergeSort

- A idéia básica do MergeSort é criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, o algoritmo MergeSort divide a sequência original em pares de dados, agrupa estes pares na ordem desejada; depois as agrupa as sequências de pares já ordenados, formando uma nova sequência ordenada de quatro elementos, e assim por diante, até ter toda a sequência ordenada.

MergeSort

- Os três passos úteis dos algoritmos dividir-para-conquistar, que se aplicam ao MergeSort são:
  - **Dividir**: Dividir os dados em subsequências pequenas; Este passo é realizado recursivamente, iniciando com a divisão do vetor de n elementos em duas metades, cada uma das metades é novamente dividida em duas novas metades e assim por diante, até que não seja mais possível a divisão (ou seja, sobrem n vetores com um elemento cada).
  - **Conquistar**: Classificar as duas metades recursivamente aplicando o mergesort;
  - **Combinar**: Juntar as duas metades em um único conjunto já classificado. Para completar a ordenação do vetor original de n elementos, faz-se o merge ou a fusão dos sub-vetores já ordenados.

http://www.youtube.com/watch?v=XaqR3G_NVoo

MergeSort

```c
void merge(int a[], int low, int high, int mid){
    int i, j, k, c[50];
    i=low;
    j=mid+1;
    k=low;
    while((i<=mid)&&(j<=high)){
        if(a[i]<a[j]){
            c[k]=a[i];
            k++;
            i++;
        }
        else{
            c[k]=a[j];
            k++;
            j++;
        }
    }
    while(i<=mid){
        c[k]=a[i];
        k++;
        i++;
    }
    while(j<=high){
        c[k]=a[j];
        k++;
        j++;
    }
    for(i=low;i<k;i++)
        a[i]=c[i];
}
```

```c
void mergeSort(int a[], int low, int high){
    int mid;
    if(low>high){
        mid=(low+high)/2;
        mergeSort(a,low,mid);
        mergeSort(a,mid+1,high);
        merge(a,low,high,mid);
    }
}
```

- Determina-se um elemento pivô. O pivô é posicionado dentro do vetor de tal forma que, todos à esquerda do pivô são menores que ele e, todos à direita do pivô são maiores. O pivô "divide"o vetor em dois subvetores. Recursivamente o quicksort é realizado na primeira metade do vetor e na segunda metade.

- Algoritmo: Seja x o vetor a ser ordenado e n o número de elementos de x. Seja a um elemento de x escolhido ao acaso (por exemplo, a=x[0]). Suponha que os elementos de x estejam divididos de tal forma que a é colocado na posição j e as seguintes condições são verdadeiras:
  - Todos os elementos nas posições de 0 a j-1 são menores que a.
  - Todos os elementos nas posições de j+1 a n-1 são maiores ou iguais a a.
- Então a está na posição correta no vetor. Se este processo for repetido para os sub-vetores x[0] a x[j-1] e x[j+1] a x[n-1], o resultado é o vetor ordenado.

http://www.youtube.com/watch?v=kDgvnbUIqT4

QuickSort

```c
int partition( int a[], int l, int r) {
    int pivot, i, j, aux;
    pivot = a[l];
    i = l;
    j = r+1;
    while(1){
        do ++i; while( a[i] <= pivot && i <= r );
        do --j; while( a[j] > pivot );
        if( i >= j ) break;
         aux = a[i];
        a[i] = a[j];
        a[j] = aux;
    }
     aux = a[l];
    a[l] = a[j];
    a[j] = aux;
    return j;
}

void quickSort( int a[], int l, int r){
    int j;
    if( l > r ){
        //dividir e conquistar
        j = partition( a, l, r);
        quickSort( a, l, j-1);
        quickSort( a, j+1, r);
    }
}
```

No main o QuickSort será chamado como **quickSort(vetor, 0, tam);**

Hash/Dispersão

- Considerar o problema de pesquisar um determinado valor em um vetor.
  - Se o vetor não está ordenado, a pesquisa requer O(n) de complexidade.
  - Se o vetor está ordenado, pode-se fazer a pesquisa binária que requer uma complexidade de O(log n).
  - Não parece haver melhor maneira de resolver o problema com custos melhorados.

Hash/Dispersão

- Ideia: Tabelas Hash/Dispersão:
  - Poderia haver uma maneira de resolver o problema em O(1).
  - Se o vetor estiver ordenado de uma determinada maneira.
- Solução?
  - Arranjar uma função mágica que, dado um determinado valor a pesquisar, nos diga exatamente a posição no vetor.
  - Esta é uma função chamada **função de dispersão/hash**.

Tabela Hash/Dispersão

- As **tabelas de hash** são um tipo de estruturação criado para o armazenamento de informação, e são uma forma extremamente simples, fácil de se implementar e, intuitiva para se organizar grandes quantidades de dados.
- Possui como ideia central a divisão do universo de dados a ser organizado em subconjuntos mais facilmente gerenciáveis.
- **Permitir armazenar e procurar rapidamente grande quantidade de dados**

- As tabelas de hash são constituídas por 2 conceitos fundamentais:
  - *Tabela de Hash*: estrutura que permite o acesso aos subconjuntos.
  - *Função de Hash*: função que realiza um mapeamento entre os valores de chaves e as entradas na tabela.

- Criar um critério simples para dividir este universo em subconjuntos com base em alguma qualidade do domínio das chaves.
  - Possuir um índice que permita encontrar o início do subconjunto certo, depois de calcular o valor de hash.
  - Isto é uma tabela de hash.

Tabela Hash/Dispersão

- Saber em qual subconjunto procurar e colocar uma chave.
  - indicar quantos subconjuntos se pretende.
  - criar uma regra de cálculo que, dada uma chave, determine em que subconjunto se deve procurar pelos dados com esta chave ou colocar estes dados (caso seja um novo elemento).
  - Isto é chamado função de hash.

Tabela Hash/Dispersão

- Gerir estes subconjuntos bem menores com um métodos simples:
  - Possuir uma estrutura ou um conjunto de estruturas de dados para os subconjuntos.
  - Existem duas filosofias: hashing fechado (ou de endereçamento aberto) ou o hashing aberto (ou encadeado).
- Alguns dos problemas que se colocam quando usamos tabelas de hash são:
  - determinar uma função de hash que minimize o número de colisões;
  - obter os mecanismos eficientes para tratar as colisões.

Funções de Hash - Exemplos

- Números x (chave) de 0 a 99 (dois dígitos)
- *tam* tamanho da tabela
- Pode-se construir uma função que coloque x no vetor em termos do algarismo das dezenas.
  f = x / tam (/ = divisão inteira)
- Ou pode-se construir construir uma função que coloque coloque x no vetor em termos do algarismo das unidades.
  f = x % tam (% = resto da divisão)

Funções de Hash - Exemplos

```c
int hash (int key, int tam) {
    return (key % tam);
}
```

- Objetivos das funções de Hash:
    - deve ser eficiente.
    - deve distribuir todos os elementos uniformemente por todas as posições da tabela.

Método normal

```
int hash (char key[], int tam) {
    int valHash = 0, k = 0;
    while (key[k] != '\0') {
        valHash = valHash + int(key[k]);
        k++;
    }
    return (valHash % tam);
}
```

Funções de Hash - Exemplos Strings

Usando a regra de Horner:

```
int hash (char key[], int tam) {
    int valHash = 0, a = 127, k = 0;
    while (key[k] != '\0') {
        valHash = valHash * a + int(key[k]);
        k++;
    }
    return (valHash % tam);
}
```

Tabela Hash/Dispersão - Colisão

- Quando a função de Hash é imperfeita poderão existir dois valores a serem colocados na mesma posição do vetor. A isto chama-se uma colisão.
- As colisões são normalmente tratadas como quem chega primeiro serve-se; isto é, o primeiro elemento a chegar a uma posição fica com ela.
- Terá de se arranjar uma solução eficiente para se determinar o que se deve fazer com o segundo valor que deveria ser colocado na mesma posição.

- O que fazer quando dois valores diferentes tentam ocupar a mesma posição?
  - Solução 1 (**hashing fechado** ou **endereçamento aberto**): procura a partir dessa posição uma posição vazia.
  - Solução 2: usar uma segunda função de hash, uma terceira, uma quarta, ?
  - Solução 3 (**hashing aberto** ou **encadeamento separado**): usa a posição do vetor como cabeça (head) de uma lista que vai conter todas as colisões dessa posição.

Hashing Fechado

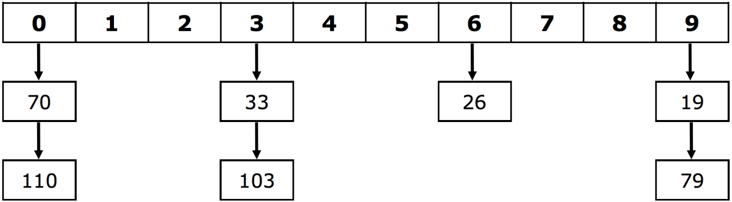
- Cria-se uma estrutura que assinale cada posição:
  - Livre (vazia e nunca ocupada)
  - Ocupada
  - Removida (vazia mas já ocupada)
- Pesquisar/remover:
  - Termina quando encontra o objeto
  - Termina quando encontra uma posição livre
  - Continua a pesquisa quando encontra uma posição removida
- Inserir (insere o objeto quando encontra):
  - uma posição assinalada como livre ou removida.

Hashing Aberto

- A estruturação dos dados é talvez a forma mais intuitiva de se implementar o conceito de Hashing.
- Consiste em ter um vetor de apontadores, com dimensão N, em que cada elemento do vetor contém uma ligação para uma lista dos elementos a guardar.
- A pesquisa pesquisa de um elemento elemento efetua-se da seguinte seguinte forma:
  - A partir de uma chave, calcular qual o elemento do vetor é a cabeça da lista que se pretende.
  - Usando um qualquer algoritmo para o efeito, pesquisar o elemento dentro daquela lista.

Exemplo 1:

- Armazenar os elementos: 19, 26, 33, 70, 79, 103 e 110.
- Usando a função de hash: hash(x) = x % 10



Exemplo 2:

- Cada elemento do vetor é um apontador para uma lista de estruturas com o mesmo valor da função de hash.
- Supondo que o vetor tem tamanho tam = 13 e que a função de hash é a que consta na tabela que se segue

Chave	A,B	C,D	E,F	G,H	I,J	K,L	M,N	O,P	Q,R	S,T	U,V	X,Y	W,Z
Hash	0	1	2	3	4	5	6	7	8	9	10	11	12

Exemplo 2:

- Armazenar os elementos: Alce, Burro, Gato, Hiena, Pato, Orca e Tigre.

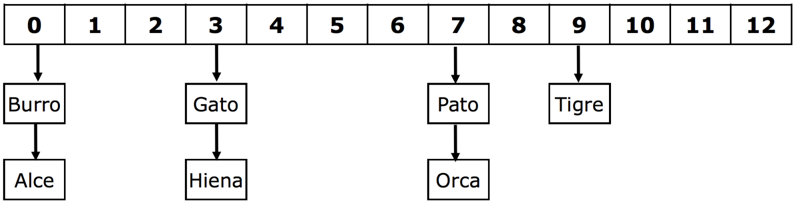


Tabela Hash/Dispersão - Hashing Aberto

- Reduz o número de comparações, quando comparado com a pesquisa sequencial.
- Necessidade de espaço em memória para armazenar o vetor de listas.

- Dispersão aberta (DA) vs. Fechada (DF)
    - Suporta a primitiva de remoção.
    - As listas de colisão não se cruzam.
    - O erro por defeito do pré-dimensionamento não é tão grave.
    - Se a tabela estiver em arquivo poupam-se muitos acessos com a sondagem linear, porque em geral os elementos que colidem estão fisicamente próximos.

- Vantagens da Dispersão:
  - A complexidade das primitivas suportadas (pesquisa, inserção e remoção na DA), no caso esperado é constante.
  - A técnica é eficiente e é só depende do fator de ocupação e da qualidade das funções (na DF).
- Problemas da Dispersão:
  - Não é uma estrutura dinâmica (o redimensionamento é necessário)
  - Não suporta primitivas que se baseiam em relações de ordem dos elementos (mínimo, máximo, percurso ordenado).
  - A complexidade das primitivas suportadas (pesquisa, inserção e remoção na DA), no pior caso é linear no nž de elementos da tabela.

O(de que??)

- Comentamos ao longo da aula sobre complexidade, algoritmo eficiente, O(n)... Mas o que significa isso?

Exercícios

- 1. Crie uma Agenda de Contatos (Nome, Telefone, Endereço) com uma Busca Binária baseada no nome do contato.
- 2. Crie uma Agenda de Contatos (Nome, Telefone, Endereço) baseada em Tabela Hash.

Na próxima aula...

Fechando Programação Estruturada/Procedural