AED1 - Aula 24
Tabelas de símbolos, implementação em vetores ordenados, árvores binárias de busca

Uma tabela de símbolos:

- Também é chamada de dicionário.
- Corresponde a um conjunto de itens,
  - em que cada item possui uma chave e um valor.
- Suporta diversas operações sobre os itens,
  - sendo busca a principal delas.
- Trata-se de um Tipo de Dado Abstrato, pois
  - o foco está no propósito da estrutura, e não em sua implementação.

Estamos interessados nas seguintes operações:

- busca - dada uma chave k, devolva um apontador
  - para um objeto com esta chave. Se não existir devolva “none”.
- min (max) - devolva um apontador
  - para um objeto com a menor (maior) chave.
- predecessor (sucessor) - dada uma chave k, devolva um apontador
  - para o objeto com a maior (menor) chave menor (maior) que k.
    - Se não existir devolva “none”.
- percurso ordenado - devolva todos os objetos
  - seguindo a ordem de suas chaves.
- seleção - dado um inteiro i, entre 1 e n, devolva um apontador
  - para o objeto com a i-ésima menor chave.
- rank - dada uma chave k, devolva o número de objetos
  - com chave menor ou igual a k.

Note que nossa definição de seleção é levemente diferente

- da adotada no problema da seleção.
- Essa escolha foi feita para que rank seja a operação inversa da seleção.

Vamos começar a pensar na implementação de uma tabela de símbolos

- e nas estruturas de dados que podemos usar para tanto.

Implementação em vetor ordenado

Considere um vetor ordenado v de tamanho n.

- Como podemos implementar as operações anteriores?

Exemplificar operações com o seguinte vetor

- 3 6 10 11 17 23 30 36

busca(8): usando busca binária temos

limite esq. → [3 6 10 (11) 17 23 30 36] ← limite dir. (meio: 11)

1º passo: 8 ≤ 11 ⇒ buscar no subvetor à esquerda

limite esq. → [3 (6) 10 11] 17 23 30 36 ← limite dir. (meio: 6)

2º passo: 8 > 6 ⇒ buscar no subvetor à direita

3 6 limite esq. → [(10) 11] ← limite dir. 17 23 30 36 (meio: 10)

3º passo: 8 ≤ 10 ⇒ buscar no subvetor à esquerda

3 6 lim. esq. → [(10)] ← lim. dir. 11 17 23 30 36

Caso parada: limite dir. = limite esq. + 1

min: [(3) 6 10 11 17 23 30 36]

predecessor(10): usando busca binária temos

[3 6 10 11 17 23 30 36] (↓ 10)

localiza o elemento

[3 (6) 10 11 17 23 30 36]

devolve o anterior, se não cair fora do vetor

percurso ordenado: 3 6 10 11 17 23 30 36

seleção(7): [3 6 10 11 17 23 (30) 36] (posições 1 2 3 4 5 6 7 8)

rank(12): usando busca binária

3 6 10 11 17 23 30 36

encontra posição em que a chave deveria estar

3 6 10 11 | 17 23 30 36

devolve número de elementos à esq. da posição, no caso 4

Biblioteca para tabela de símbolos

Segue o código da interface TSvetorOrdenado.h:

```c
typedef struct ts TS;
typedef int Chave;
typedef int Cont;

typedef struct item
{
    Chave chave;
    Cont conteudo;
} Item;

TS *TScria(Item *v, int n);
Item *TSbusca(TS *tab, Chave x);
Item *TSmin(TS *tab);
Item *TSmax(TS *tab);
Item *TSpred(TS *tab, Chave x);
Item *TSsuc(TS *tab, Chave x);
void TSperc(TS *tab);
Item *TSselec(TS *tab, int i);
int TSrank(TS *tab, Chave x);
```

Implementação da biblioteca em vetor ordenado

A seguir temos os códigos da biblioteca usando vetor.

```c
#include <stdio.h>
#include <stdlib.h>

#include "TSvetorOrdenado.h"

struct ts
{
    Item *v;
    int n;
};
```

Busca binária: observe que a seguinte implementação da buscaBinaria

- devolve uma posição, ainda que não encontre a chave buscada.
    - Que posição é essa?

```c
int buscaBinaria(Item v[], int n, Chave x) {
    int e, m, d;
    e = -1;
    d = n;
    while (e < d - 1)
    {
        m = (e + d) / 2;
        if (v[m].chave < x)
            e = m;
        else
            d = m;
    }
    return d;
}
```

- Invariante e corretude:
    - No início de cada iteração do laço temos
        - v[e] < x <= v[d].
    - Na primeira iteração isso vale pois
        - v[-1] e v[n] não estão definidos.
    - Quando o algoritmo sai do laço temos e = d - 1.
        - Assim, v[e] = v[d - 1] < x <= v[d].

        - Portanto, ao devolver d o algoritmo está indicando
            - a posição que a chave x deve ocupar no vetor,
                - quer ela esteja nele ou não.
- Eficiência: leva tempo O(log n) no pior caso.

Inicializa:
```
TS *TScria(Item v[], int n) {
    int i;
    TS *tab;
    tab = (TS *)malloc(sizeof(TS));
    tab->v = (Item *)malloc(n * sizeof(Item));
    tab->n = n;
    for (i = 0; i < n; i++)
        tab->v[i] = v[i];
    insertionSort(tab->v, tab->n);
    return tab;
}
```

Busca:
```
Item *TSbusca(TS *tab, Chave x) {
    int i;
    i = buscaBinaria(tab->v, tab->n, x);
    if (tab->v[i].chave == x)
        return &(tab->v[i]);
    return NULL;
}
```

Mínimo:
```
Item *TSmin(TS *tab) {
    return &(tab->v[0]);
}
```

Predecessor:
```
Item *TSpred(TS *tab, Chave x) {
    int i;
    i = buscaBinaria(tab->v, tab->n, x);
    if (tab->v[i].chave == x && i != 0)
```

```
        return &(tab->v[i - 1]);
    return NULL;
}
```

Percurso ordenado:

```
void TSperc(TS *tab) {
    int i;
    for (i = 0; i < tab->n; i++)
        printf("(%d, %d) ", tab->v[i].chave, tab->v[i].conteudo);
    printf("\n");
}
```

Seleção:

```
Item *TSselec(TS *tab, int pos) {
    return &(tab->v[pos - 1]); // vetor começa em 0
}
```

Rank:

```
int TSrank(TS *tab, Chave x) {
    int i, pos;
    i = buscaBinaria(tab->v, tab->n, x);
    if (tab->v[i].chave == x)
        pos = i + 1;
    else
        pos = i;
    return pos;
}
```

Eficiência das operações:

- busca - O(log n), deriva da busca binária.
- min (max) - O(1).
- predecessor (sucessor) - O(log n), deriva da busca binária.
- percurso ordenado - O(n), mínimo possível já que é o tamanho da saída.
- seleção - O(1).
- rank - O(log n), deriva da busca binária.

Mas vetor ordenado não é eficiente quando o conjunto de itens é dinâmico.

- inserção - O(n). Por que?

- remoção - O(n). Por que?

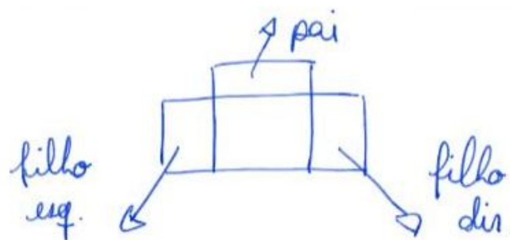
Árvores de busca são alternativa para implementar

- tabelas de símbolos que trabalham com conjuntos dinâmicos.
    - Em particular, estamos interessados em árvores binárias de busca.

Árvores binárias de busca

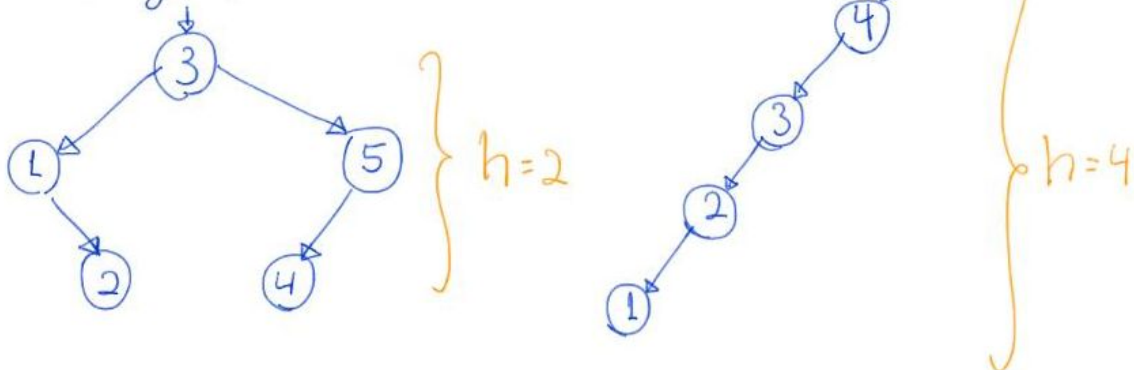
Relembrando, cada nó de uma árvore binária corresponde a um objeto com:

- uma chave
- um apontador para o filho esquerdo
- um apontador para o filho direito
- um apontador para o pai



Exemplos de árvores binárias distintas,

- mas que contém o mesmo conjunto de objetos.



A raiz de uma árvore é o único nó que não é filho de outro.

- Nas árvores anteriores as raízes são 3 e 5, respectivamente.

Uma folha é um nó que não tem filhos.

- Nas árvores anteriores as folhas são 2, 4 e 1, respectivamente.

A altura (h) de uma árvore é o comprimento do maior caminho da raiz até uma folha.

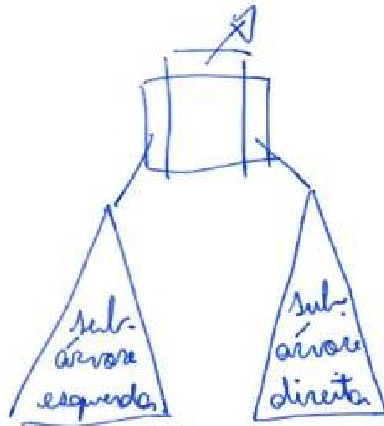
- Nas árvores anteriores as alturas são 2 e 4, respectivamente.

De fato, a altura de uma árvore binária com n nós pode variar muito:

- desde ~= lg n, caso seja perfeitamente balanceada,
- até n-1, caso seja uma lista encadeada.

Podemos definir uma árvore binária recursivamente como sendo:

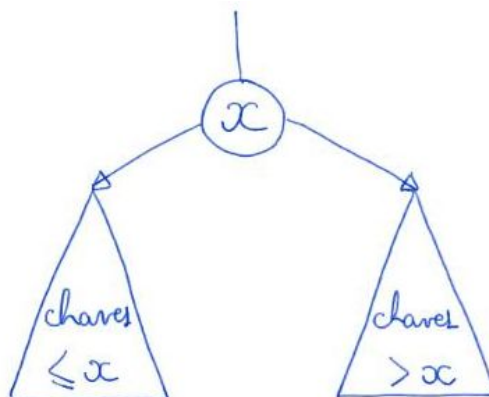
- um nó com uma subárvore esquerda e uma subárvore direita,
- ou uma árvore vazia.



Essa definição recursiva vai nos ajudar a pensar nas operações.

O que diferencia uma árvore binária qualquer de uma árvore binária de busca

- é a propriedade de busca, i.e., dado um nó com chave x:
    - os elementos na subárvore esquerda tem chave <= x
    - e os objetos na subárvore direita tem chave > x.



- Observe que esta propriedade mantém os elementos ordenados na árvore.

Uma importante aplicação de árvores binárias de busca

- é na implementação de tabelas de símbolos dinâmicas.
    - Segue o código da interface TSarvoreBinaria.h:

```
typedef int Chave;
typedef int Cont;
```

```c
typedef struct noh
{
    Chave chave;
    Cont conteudo;
    int tam;
    struct noh *pai;
    struct noh *esq;
    struct noh *dir;
} Noh;

typedef Noh TS;

Noh *TSbusca(TS *tab, Chave x);
Noh *TSmin(TS *tab);
Noh *TSmax(TS *tab);
Noh *TSpred(TS *tab, Chave x);
Noh *TSsuc(TS *tab, Chave x);
void TSperc(TS *tab);
Noh *TSselec(TS *tab, int i);
int TSrank(TS *tab, Chave x);
TS *TSinserir(TS *tab, Chave chave, Cont conteudo);
TS *TSremover(TS *tab, Chave chave);
```

Agora vamos discutir como implementar as operações da tabela de símbolos

- numa árvore binária de busca e vamos avaliar a eficiência das mesmas
    - em função da altura (h) da árvore.
- Atentem que diversas operações não utilizam o campo pai.

Busca(k):

- Comece na raiz,
- repita o seguinte processo até chegar num apontador vazio
    - se a chave do nó atual = k devolva apontador para ele
    - se k < chave do nó atual desça para o filho esquerdo
    - se k > chave do nó atual desça para o filho direito
- devolva “none”.

Versão recursiva

```c
Noh *TSbuscaR(Arvore r, Chave chave) {
    if (r == NULL)
        return r;
    if (r->chave == chave)
        return r;
    if (chave < r->chave)
        return TSbuscaR(r->esq, chave);
    // r->chave < chave
    return TSbuscaR(r->dir, chave);
}
```

Versão iterativa

```c
Noh *TSbusca(Arvore r, Chave chave) {
    while (r != NULL && r->chave != chave)
    {
        if (chave < r->chave)
            r = r->esq;
        else
            r = r->dir;
    }
    return r;
}
```

- Eficiência: ambas levam tempo O(altura) no pior caso,
    - já que em cada chamada recursiva (ou iteração)
        - descem um nível na árvore.

Min (max):

- Comece na raiz,
- desça pelo filho esquerdo (direito) até encontrar um apontador vazio.
- Devolva um apontador para o último objeto visitado.

```c
// supõe que r não é vazia
Noh *TSmin(Arvore r) {
    while (r->esq != NULL)
        r = r->esq;
    return r;
}
```

- Eficiência: leva tempo O(altura) no pior caso,
    - já que em cada iteração descem um nível na árvore.

Percurso ordenado:

- Se árvore corrente não for vazia
  - chame recursivamente “percurso ordenado”
    - para subárvore enraizada no filho esquerdo
  - devolva objeto da raíz
  - chame recursivamente “percurso ordenado”
    - para subárvore enraizada no filho direito

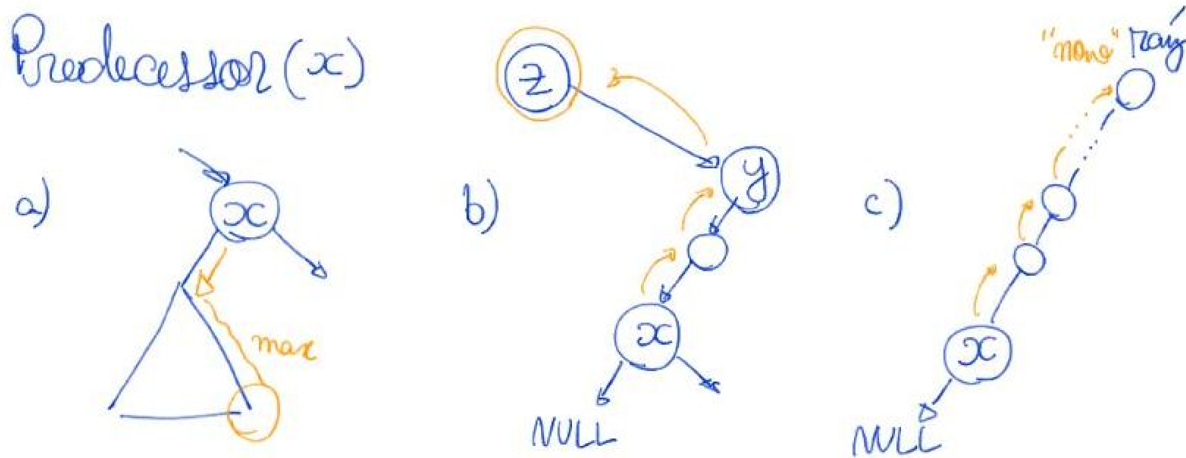
```
void TSperc(TS *tab) {
    inOrdemR(tab);
    printf("\n");
}

void inOrdemR(Arvore r) {
    if (r != NULL)
    {
        inOrdemR(r->esq);
        printf("(%d, %d) ", r->chave, r->tam);
        inOrdemR(r->dir);
    }
}
```

- Como vimos na aula sobre árvores binárias,
  - esta forma de varredura de árvore é chamada de inordem
    - pois a raiz de cada subárvore é visitada entre os filhos.
- Também é conhecida por e-r-d, referência à esquerda-raiz-direita,
  - e pode ser implementada usando uma pilha explícita.
- Eficiência: leva tempo O(n), pois visita cada nó uma vez.

Predecessor (sucessor):

- Para implementar essas operações, vamos usar em cada nó,
  - um apontador para o pai do mesmo,
    - sendo que o pai da raiz é NULL.
- Isso nos obriga a atualizar esses valores
  - nas operações que alteram a árvore, i.e., inserção e remoção.
- Procedimento:
  - Encontre o objeto alvo usando busca.
  - a) se o filho esquerdo (direito) é não vazio,
    - devolva max da subárvore enraizada neste filho.
  - b) caso contrário, siga repetidamente apontadores
    - para o antecessor de x até visitar nós y e z tal que

- y é filho direito (esquerdo) de z.
  - Devolva z.
- Observe que x é o min (max) da subárvore direita de z.
  - Portanto, x é sucessor (predecessor) de z.
- c) se não encontrar, devolva “none”.




```
Noh *TSpred(TS *tab, Chave x) {
    Noh *q, *p;
    q = TSbusca(tab, x);
    if (q == NULL)
        return NULL;
    if (q->esq != NULL)
        return TSmax(q->esq);
    p = q->pai;
    while (p != NULL && p->esq == q)
    {
        q = p;
        p = p->pai;
    }
    return p;
}
```

- Eficiência: leva tempo O(altura),
  - pois a busca é O(altura), máximo é O(altura) e
    - no segundo laço em cada iteração sobe um nível na árvore.

Quiz1: Suponha que os apontadores pai estão vazios em uma árvore.

- Como projetar uma função para preenchê-los?

Quiz2: Também é possível modificar a função para encontrar o predecessor,

- sem precisar do campo pai. Como fazer isso?
    - Dica: Envolve modificar a busca, para que ela guarde
        - o antepassado mais recente do nó com chave menor que ele.

Compilando biblioteca

Para implementar e compilar um programa que usa nossa biblioteca,

- primeiro incluímos uma chamada para ela no início do programa,

```
#include "TS.h"
```

- então compilamos a biblioteca em um programa objeto
  "gcc -c TS.c" ou
  "gcc -Wall -O2 -pedantic -Wno-unused-result -c TS.c"
- e, finalmente, compilamos o programa principal usando esse programa objeto
  "gcc TS.o usaTS.c -o usaTS" ou
  "gcc -Wall -O2 -pedantic -Wno-unused-result TS.o usaTS.c -o usaTS"

Também podemos compilar o programa principal em um programa objeto
  "gcc -c usaTS.c" ou
  "gcc -Wall -O2 -pedantic -Wno-unused-result -c usaTS.c"

- e então compilar os dois programas objetos no executável
  "gcc TS.o usaTS.o -o usaTS"

Ou, no extremo oposto, compilar tudo diretamente, sem usar programas objeto
  "gcc TS.c usaTS.c -o usaTS" ou
  "gcc -Wall -O2 -pedantic -Wno-unused-result TS.c usaTS.c -o usaTS"