AED1 - Aula 20
Tabelas de símbolos, implementação em vetores ordenados, árvores binárias

Tabelas de símbolos

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.

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

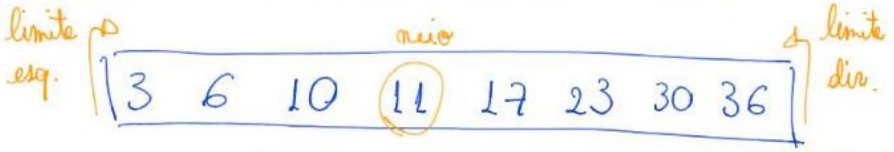
Considere um vetor ordenado v de tamanho n.

- Como podemos implementar as operações anteriores?

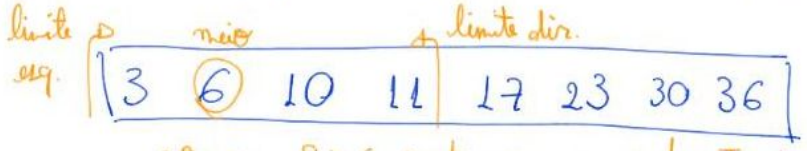
Exemplificar operações com o seguinte vetor

- 3 6 10 11 17 23 30 36

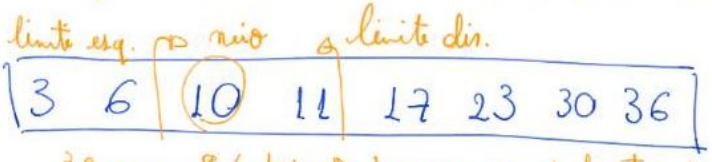
busca(8): usando busca binária temos



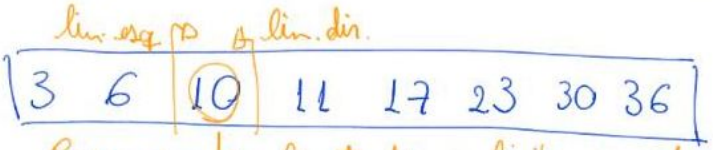
1° passo: $8 \leq 11 \Rightarrow$ buscar no subvetor à esquerda



2° passo: $8 > 6 \Rightarrow$ buscar no subvetor à direita

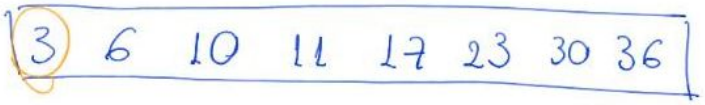


3° passo: $8 \leq 10 \Rightarrow$ buscar no subvetor à esquerda

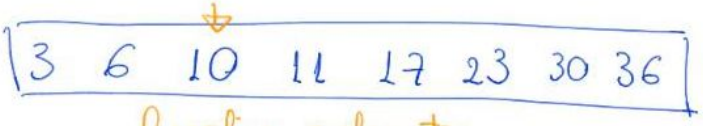


Caso parada: limite dir. = limite esq. + 1

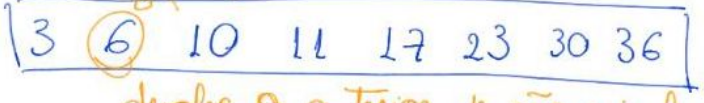
min:



predecessor(10): usando busca binária temos



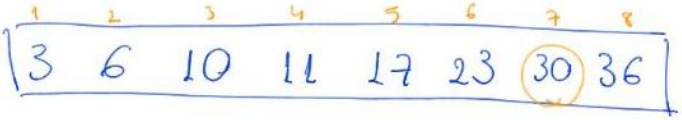
localiza o elemento

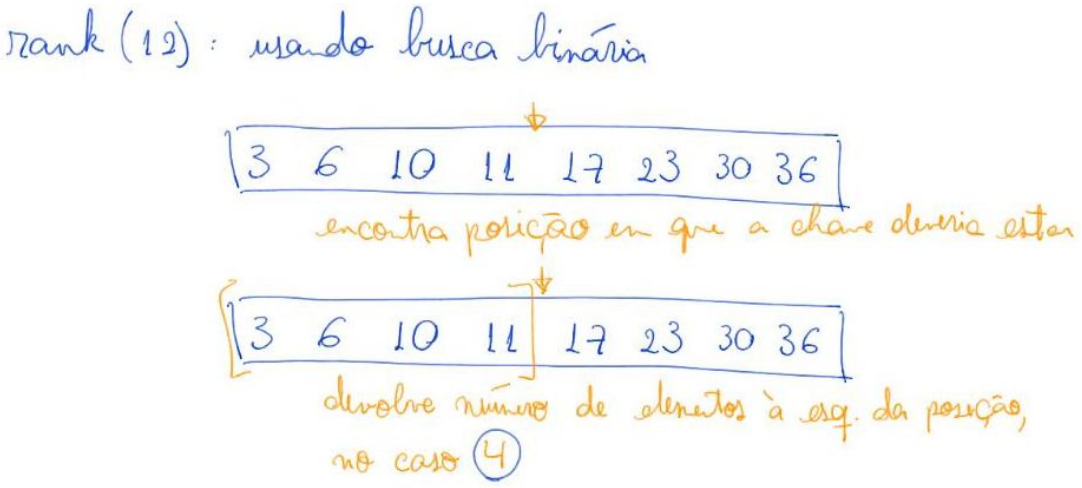


devolve o anterior, se não cair fora do vetor

percurso ordenado: 3 6 10 11 17 23 30 36

seleção(7):





Códigos das operações:

- busca binária
    - observe que a seguinte implementação da buscaBinaria
        - devolve uma posição, ainda que não encontre a chave buscada.
            - Que posição é essa?

```
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.
- busca

```
int *busca(int v[], int n, int x)
{
```

```
    int i;
    i = buscaBinaria(v, n, x);
    if (v[i] == x)
        return &v[i];
    return NULL;
}
```

- min

```
int *min(int v[], int n)
{
    return &v[0];
}
```

- predecessor

```
int *pred(int v[], int n, int x)
{
    int i;
    i = buscaBinaria(v, n, x);
    if (v[i] == x && i != 0)
        return &v[i - 1];
    return NULL;
}
```

- percurso ordenado

```
void perc(int v[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
}
```

- seleção

```
int *selec(int v[], int n, int i)
{
    return &v[i - 1];
}
```

- rank

```
int rank(int v[], int n, int x)
{
    int i;
    i = buscaBinaria(v, n, x);
    if (v[i] == x)
        i++;
    return i;
}
```

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 funciona/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.
- Mas, antes de chegarmos lá, vamos estudar o que são essas árvores,
    - que generalizam listas encadeadas.
- Em particular, estamos interessados em árvores binárias.

Biblioteca para tabela de símbolos implementada em vetor ordenado

Segue o código da interface fila.h:

```
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);
```

A seguir temos a implementação da biblioteca usando vetor.

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

#include "TS.h"

struct ts
```

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

void insertionSort(Item v[], int n)
{
   int i, j;
   Item aux;
   for (j = 1; j < n; j++)
   {
       aux = v[j];
       for (i = j - 1; i >= 0 && aux.chave < v[i].chave; i--)
           v[i + 1] = v[i];
       v[i + 1] = aux; /* por que i+1? */
   }
}
```

Extra: observe que a seguinte implementação do insertionSort

- é particularmente eficiente.
    - Por que?

```
void insertionSortImproved(Item v[], int n)
{
   int i, j;
   Item aux;
   j = n - 1;
   for (i = n - 1; i > 0; i--)
       if (v[i - 1].chave > v[i].chave)
       {
           aux = v[i];
           v[i] = v[i - 1];
           v[i - 1] = aux;
           j = i;
       }
   for (j++; j < n; j++)
   {
       aux = v[j];
       for (i = j - 1; aux.chave < v[i].chave; i--)
           v[i + 1] = v[i];
       v[i + 1] = aux; /* por que i+1? */
   }
}

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;
}

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];
    insertionSortImproved(tab->v, tab->n);
    return tab;
}

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;
}

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

Item *TSmax(TS *tab)
{
    return &(tab->v[tab->n - 1]);
}

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;
}

Item *TSsuc(TS *tab, Chave x)
{
    int i;
    i = buscaBinaria(tab->v, tab->n, x);
    if (tab->v[i].chave == x && i != tab->n - 1)
        return &(tab->v[i + 1]);
    return NULL;
}

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");
}

Item *TSselec(TS *tab, int i)
{
    return &(tab->v[i - 1]);
}

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

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”

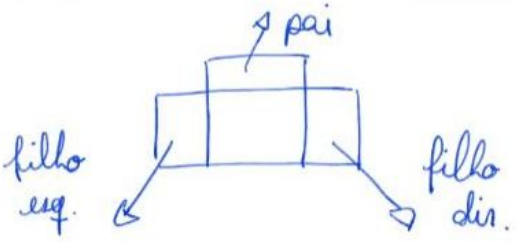
Também podemos compilar o programa principal em um programa objeto
“gcc -c usaTS.c” ou
“gcc -Wall -O2 -pedantic -Wno-unused-result -c usaTS.c”

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

Árvores binárias

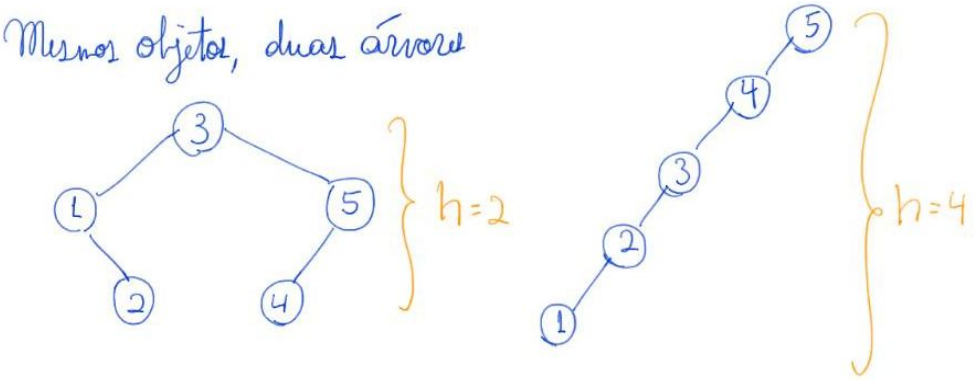
Cada nó de uma árvore binária corresponde a um objeto com:

- 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.

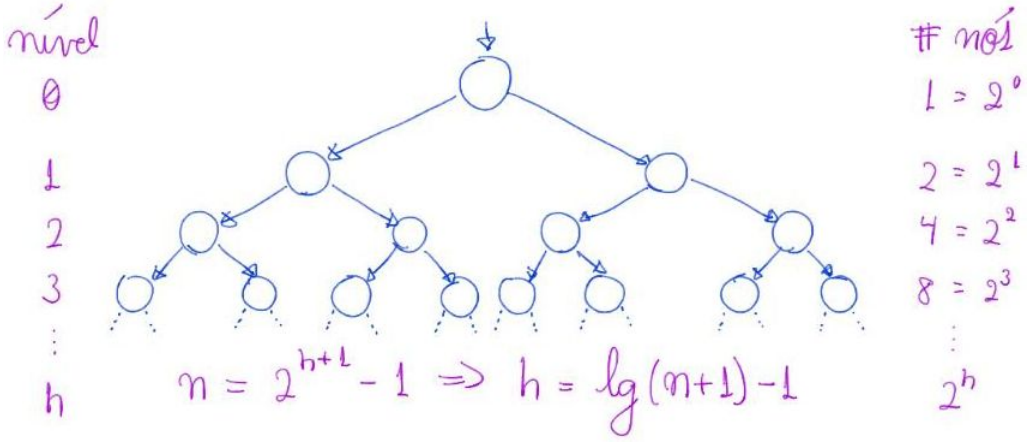
- Nas árvores anteriores as folhas são 2, 4 e 1, respectivamente.

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:

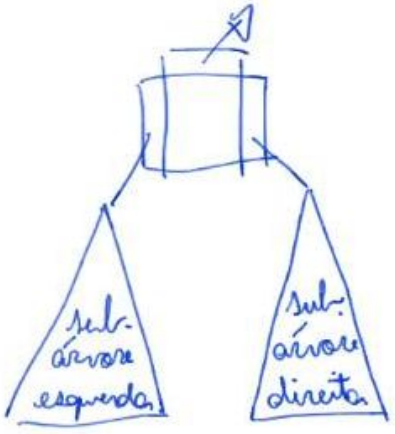
- desde ~= lg n, caso seja perfeitamente balanceada,



- 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.