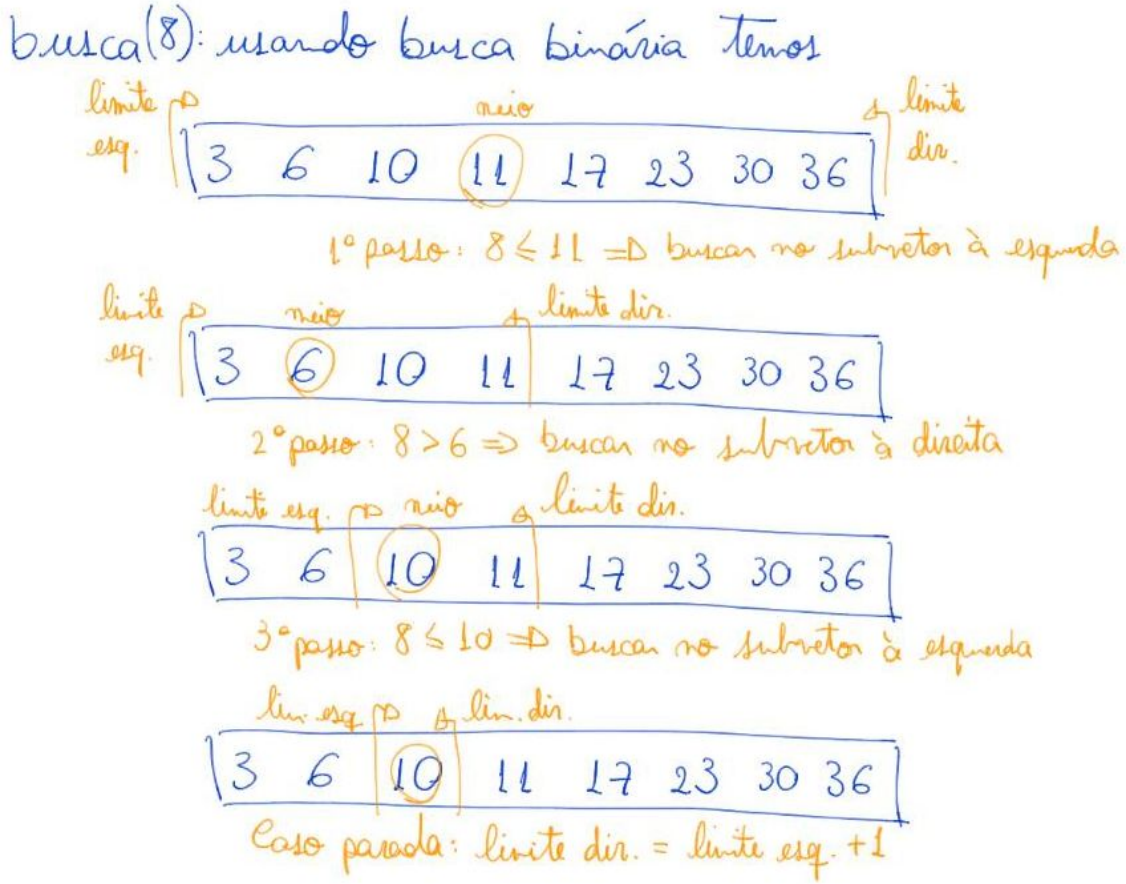
AED2 - Aula 04
Vetores ordenados e árvores de busca

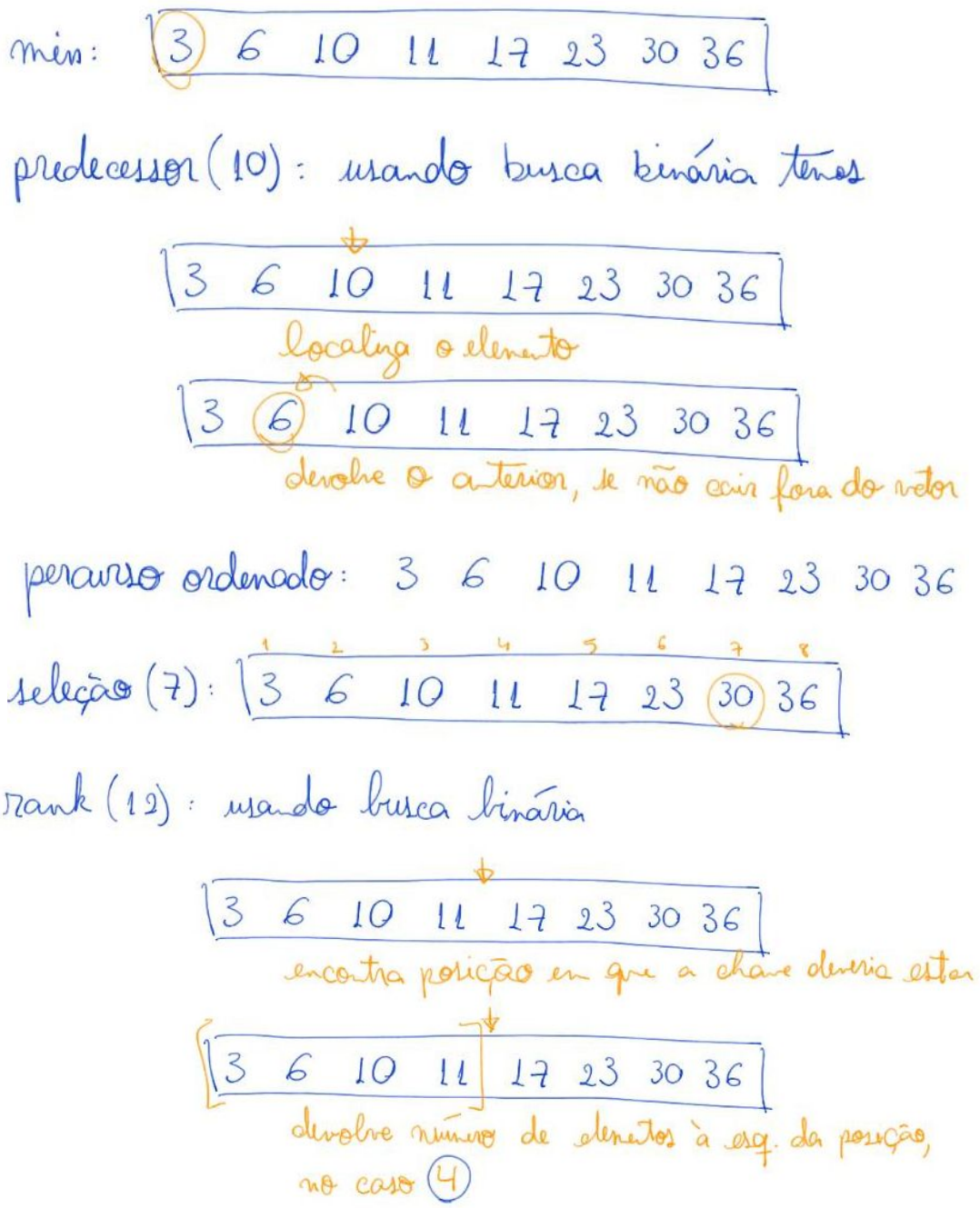
Considere um vetor ordenado v de tamanho n. Ele suporta as seguintes operações:

- 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.

Exemplificar operações com o seguinte vetor

- 3 6 10 11 17 23 30 36





Códigos das operações:

- busca binária

```
int buscaBinariaR(int v[], int e, int d, int x)
{
   int m;
   if (e == d - 1)
       return d;
   m = (e + d) / 2;
   if (v[m] < x)
       return buscaBinariaR(v, m, d, x);
   return buscaBinariaR(v, e, m, x);
}
```

- busca

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

- min

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

- predecessor

```
void *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

```
void *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 balanceadas de busca são alternativa para lidar com conjuntos dinâmicos

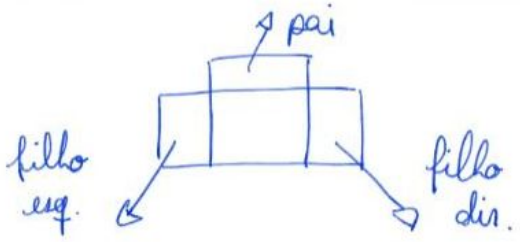
- busca - O(log n).
- min (max) - O(log n).
- predecessor (sucessor) - O(log n).
- percurso ordenado - O(n).
- seleção - O(log n).
- rank - O(log n).
- inserção - O(log n).
- remoção - O(log n).

Antes de entrarmos em detalhes sobre como manter uma árvore de busca balanceada (ou mesmo nos motivos para estarmos interessados nisso), vamos tratar da implementação das operações em árvores binárias de busca.

Árvores binárias de busca

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



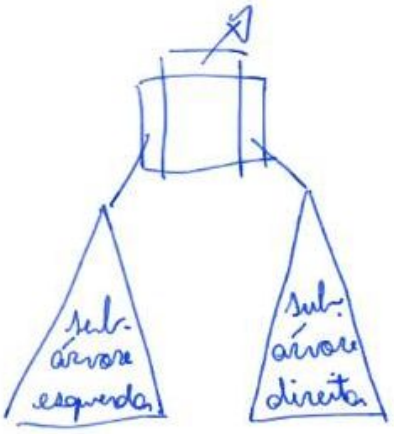

```
typedef int Item;
typedef int Chave;
```

```
typedef struct noh
{
   Chave chave;
   Item conteudo;
   struct noh *pai;
   struct noh *esq;
   struct noh *dir;
} Noh;

typedef Noh *Arvore;
```

Na definição recursiva uma árvore binária é descrita como:

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

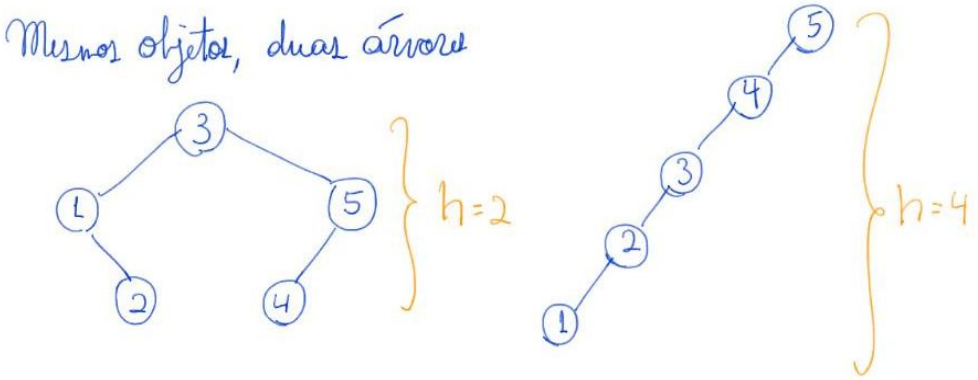


Essa definição recursiva vai nos ajudar a pensar nas operações.

O que diferencia uma árvore binária qualquer de uma árvore binária de busca é a propriedade de busca, i.e., para todo objeto x:

- os objetos na subárvore esquerda de x tem chave <= que a chave de x,
- os objetos na subárvore direita de x tem chave > que a chave de x.

Vale notar que, dado um conjunto de objetos, mais de uma árvore pode armazená-lo. Por exemplo:



Em particular, as árvore do exemplo anterior não apenas são diferentes mas também possuem alturas (h) diferentes. De fato a altura de uma árvore binária com n nós pode varia muito:

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

Agora vamos discutir como implementar as operações numa árvore binária de busca e vamos avaliar a eficiência das mesmas em função da altura (h) da árvore:

- busca(k) - com eficiência O(altura)
    - 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”

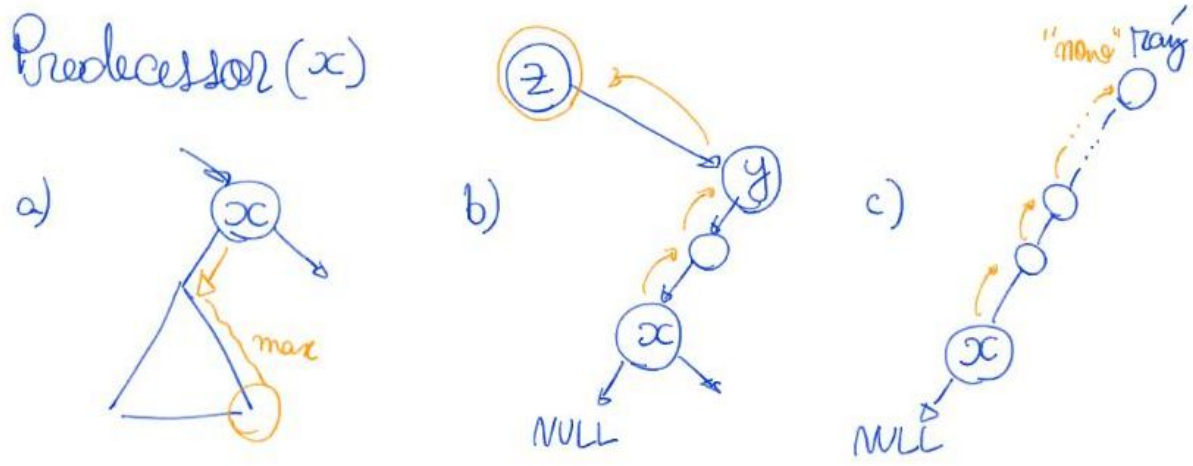
```
Noh *buscaI(Arvore r, Chave chave)
{
   while (r != NULL && r->chave != chave)
   {
       if (chave < r->chave)
           r = r->esq;
       else
           r = r->dir;
   }
   return r;
}
```

- min (max) - com eficiência O(altura)
    - comece na raiz
    - desça pelo filho esquerdo (direito) até encontrar um apontador vazio
    - devolva um apontador para o último objeto visitado

```
Noh *min(Arvore r)
{
   while (r->esq != NULL)
       r = r->esq;
   return r;
}
```

- predecessor (sucessor) - com eficiência O(altura)
    - 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 *predecessor(Noh *x)
{
   Noh *p;
   if (x->esq != NULL)
       return max(x->esq);
   p = x->pai;
   while (p != NULL && p->esq == x)
   {
       x = p;
       p = p->pai;
   }
   return p;
}
```

- percurso ordenado - com eficiência O(n)
  - 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

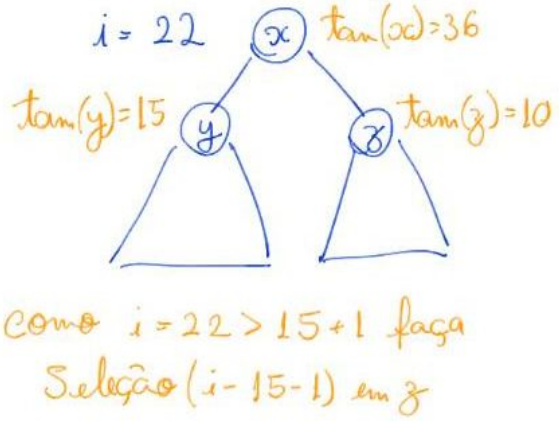
```
void inOrdem(Arvore r)
{
   if (r != NULL)
   {
       inOrdem(r->esq);
       printf("%d ", r->chave);
       inOrdem(r->dir);
   }
}
```

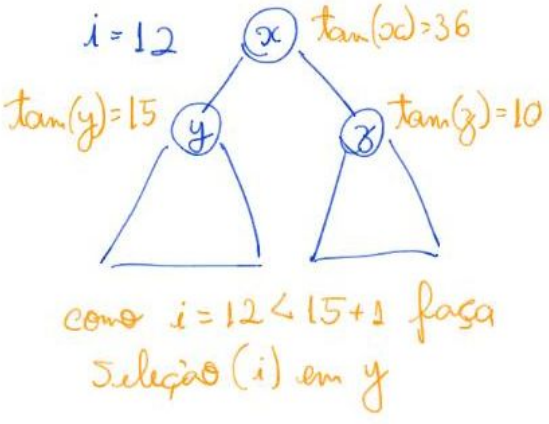
- seleção(i) - com eficiência O(altura)
  - para ficar eficiente é necessário armazenar, em cada nó, o número de objetos (tam) na árvore enraizada neste objeto.

        - isso nos obriga a atualizar esses valores nas operações que alteram a árvore, i.e., inserção e remoção.

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

    - note que, dada uma árvore com raíz x, filho esquerdo y e filho direito z, temos a relação:
        - tam(x) = tam(y) + tam(z) + 1
    - comece na raiz
        - seja tam_fe = tam(filho esquerdo)
        - se i = tam_fe + 1 devolva um apontador para a raiz
        - se i < tam_fe + 1 chame “selecao(i)” recursivamente na subárvore esquerda
        - se i > tam_fe + 1 chame “selecao(i - tam_fe - 1) recursivamente na subárvore direita






```
Noh *selecao(Arvore r, int i)
{
   int t_esq;
   if (r == NULL)
       return NULL;
   if (r->esq != NULL)
       t_esq = r->esq->tam;
   else
       t_esq = 0;
   if (i == t_esq + 1)
       return r;
   if (i < t_esq + 1)
       return selecao(r->esq, i);
   // i > t_esq + 1
```

```
    return selecao(r->dir, i - t_esq - 1);
}
```

- rank(k) - com eficiência O(altura)
    - assim como no caso da seleção, para ficar eficiente é necessário armazenar, em cada nó, o número de objetos (tam) na árvore enraizada neste objeto.
        - lembrar de atualizar o valor de tam nas operações que modificam a árvore.
    - note que, o rank de uma chave k corresponde ao número de objetos com chave menor ou igual a k.
        - por isso a ideia é fazer uma busca em que vamos somando o número de nós que ficou à esquerda do caminho percorrido.
    - comece na raiz, com uma variável rank = 0.
    - repita o seguinte processo até chegar num apontador vazio
        - se k < chave do nó atual desça para o filho esquerdo
        - caso contrário
            - rank += tam(filho esquerdo) + 1
            - se a chave do nó atual = k então devolva rank
            - se k > chave do nó atual então desça para o filho direito
    - devolva rank

```
int rank(Arvore r, Chave chave)
{
    int tam = 0, t_esq;
    while (r != NULL && r->chave != chave)
    {
        if (chave < r->chave)
            r = r->esq;
        else
        {
            if (r->esq != NULL)
                t_esq = r->esq->tam;
            else
                t_esq = 0;
            tam += t_esq + 1;
            r = r->dir;
        }
    }
    if (r != NULL)
    {
        if (r->esq != NULL)
            tam += r->esq->tam;
        tam++;
    }
    return tam;
}
```