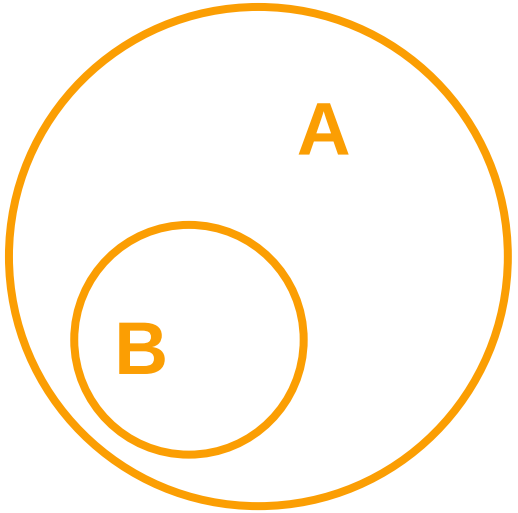
CONJUNTO E MULTICONJUNTO

Prof. André Backes | @progdescomplicada

CONJUNTO

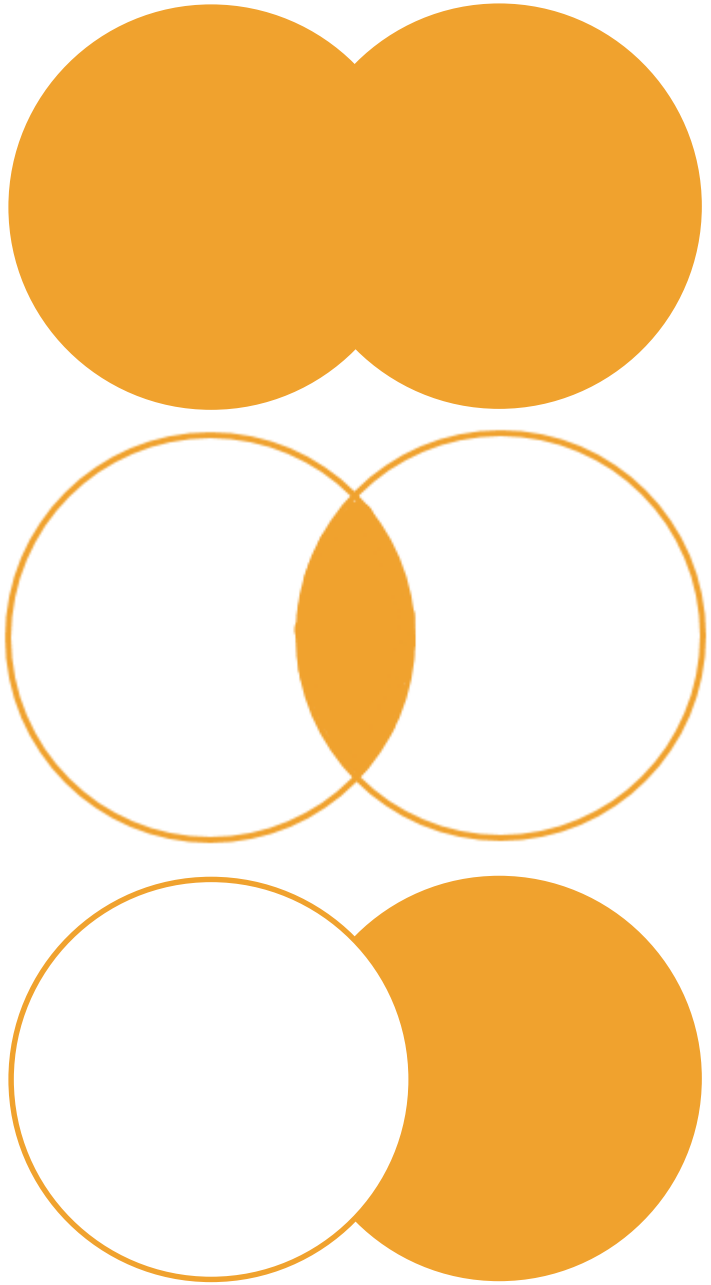
Conjunto | Definição

- Em Ciência da Computação, um conjunto é uma estrutura de dados utilizada para armazenar e organizar dados, sem repetições e sem qualquer ordem particular



Conjunto | Definição

- Sua implementação segue o conceito matemático de um conjunto finito
- Suporta operações matemáticas sobre conjuntos
  - União
  - Intersecção
  - Diferença



Conjunto | Aplicações

- Controle de permissões em sistemas
  - Em sistemas de autenticação, conjuntos podem ser usados para armazenar permissões de usuários
- Eliminação de duplicatas
  - Em algoritmos de filtragem de dados, como busca por resultados únicos em listas ou bancos de dados
- Tags em sistemas de categorização
  - Em sistemas que permitem a categorização de conteúdo (como em blogs ou redes sociais), conjuntos são usados para armazenar *tags* (rótulos)

Conjunto | TAD

- Um conjunto pode ser implementado usando
  - Tabela Hash
  - Árvore Binária de Busca
- Além disso, ele pode ser ordenado ou não

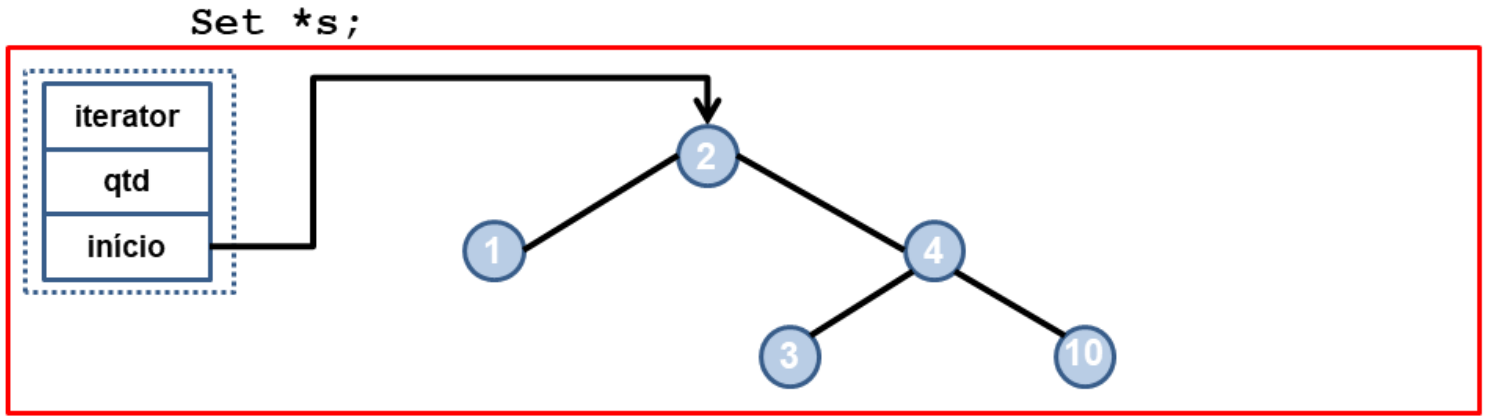
Conjunto | TAD

- Utiliza uma AVL para armazenar os elementos
  - Vantagem: reuso de estruturas prontas, custo máximo das operações é sempre *O(log N)*, menor uso de memória do que hash
  - Desvantagem: maior complexidade de implementação, desempenho inferior a hash
- O *iterator* é uma simples lista dinâmica
  - Será definido mais tarde

```
//Definição do tipo Set
struct set{
    ArvAVL* arv;
    int qtd;
    struct iterator *iter;
};

typedef struct set Set;
```

Conjunto | TAD

- O conjunto é mantido usando três campos
  - início
  - qtd
  - Um ponteiro para criar um *iterator*



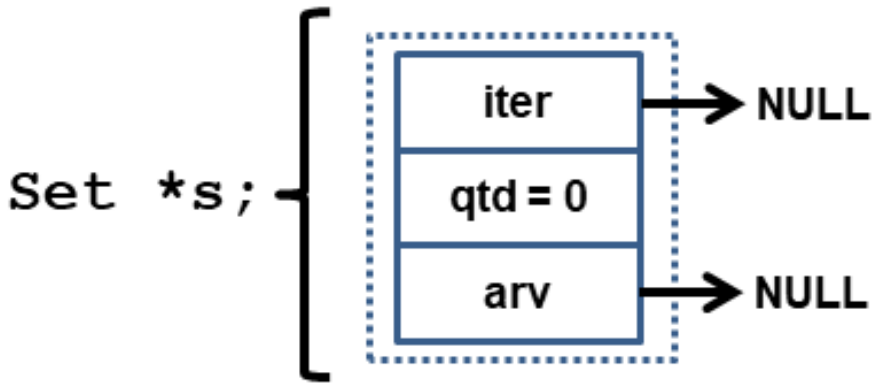
Conjunto | Criando e destruindo

- Criação
  - Aloca uma área de memória para o conjunto
  - Corresponde a memória necessária para armazenar a estrutura do conjunto
  - Inicializa os demais campos

```
Set* criaSet(){
    Set* s = (Set*) malloc(sizeof(Set));
    if(s != NULL){
        s->arv = cria_ArvAVL();
        s->qtd = 0;
        s->iter = NULL;
    }
    return s;
}
```

Conjunto | Criando e destruindo

- Ao final desse processo temos um conjunto vazio
- Podemos verificar se o conjunto está vazio pelos campos
  - qtd é igual a 0?
  - arv é igual a NULL?



Conjunto | Criando e destruindo

- Liberação é feita em três etapas
  - Liberar a memória alocada para a AVL
  - Liberar a memória alocada para a lista dinâmica
  - Liberar a memória alocada para a estrutura conjunto

```
void liberaSet(Set* s){
    if(s != NULL){
        libera_ArvAVL(s->arv);

        struct iterator* no;
        while(s->iter != NULL){
            no = s->iter;
            s->iter = s->iter->prox;
            free(no);
        }

        free(s);
    }
}
```

Conjunto | Inserção

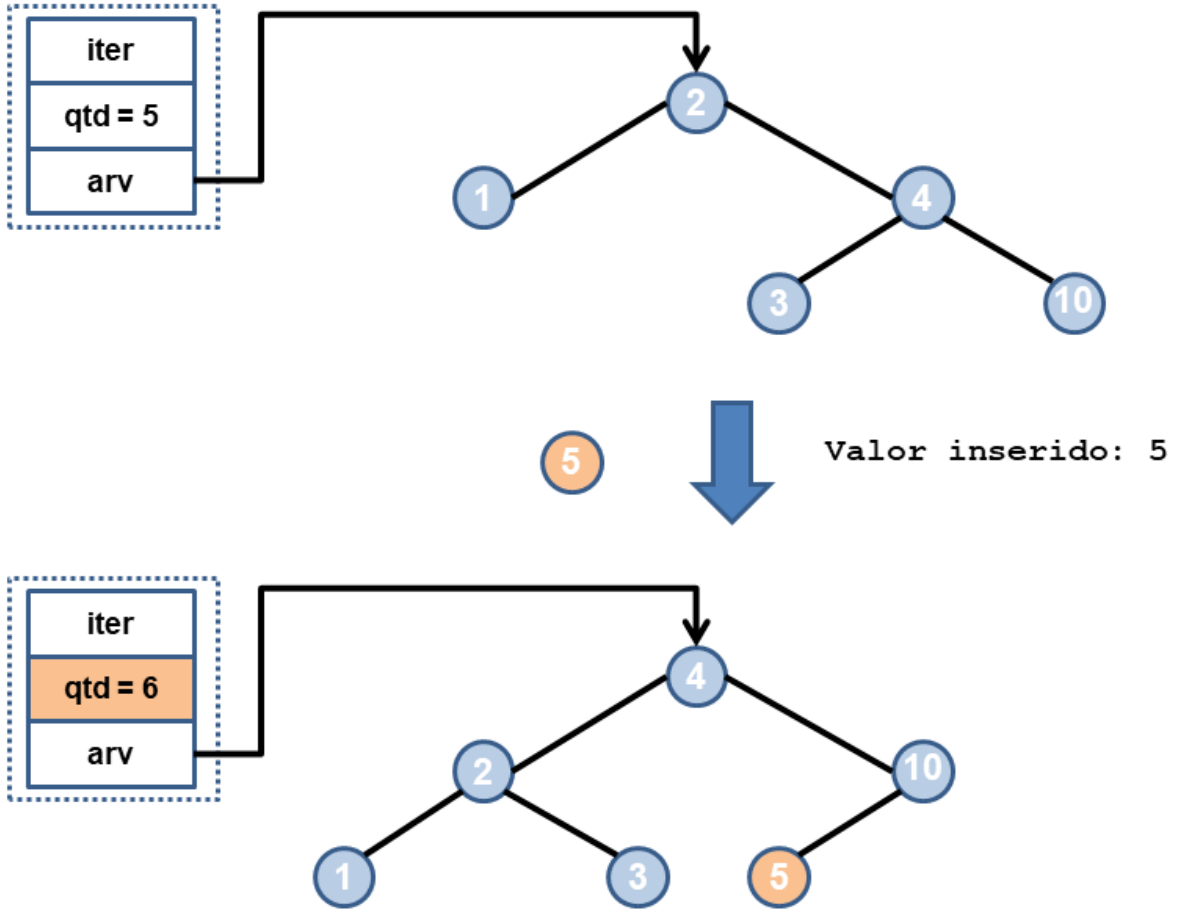
- A tarefa de inserir um novo elemento em um conjunto é bastante simples

- Basta chamar a função de inserção da árvore AVL

```
int insereSet(Set* s, int num){
    if(s == NULL)
        return 0;

    if(insere_ArvAVL(s->arv,num)){
        s->qtd++;
        return 1;
    }else
        return 0;
}
```

Conjunto | Inserção

- Ao reutilizar a implementação da AVL, não precisamos nos preocupar com a eficiência do conjunto



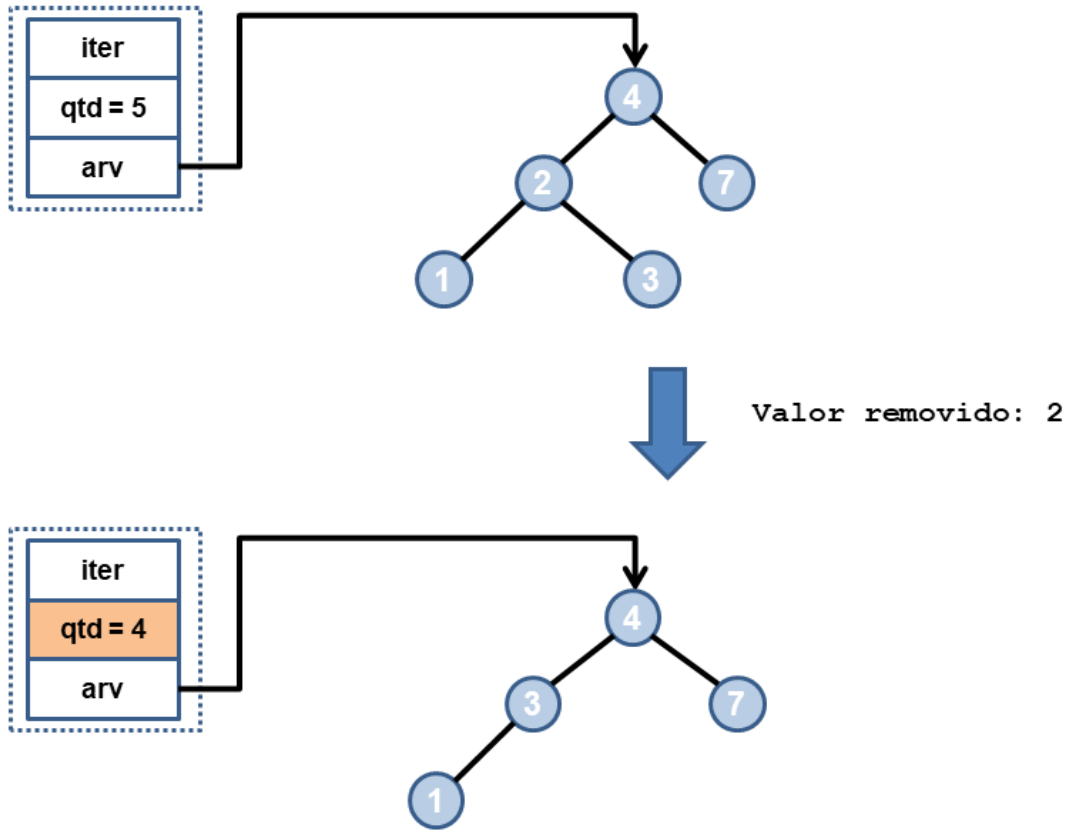
Conjunto | Remoção

- Assim, como a inserção, a remoção consiste, basicamente, em chamar a função de remoção da árvore AVL

```
int removeSet(Set* s, int num){
    if(s == NULL)
        return 0;

    if(remove_ArvAVL(s->arv,num)){
        s->qtd--;
        return 1;
    }else
        return 0;
}
```

Conjunto | Remoção

- Ao reutilizar a implementação da AVL, não precisamos nos preocupar com a eficiência do conjunto



Conjunto | Busca

- A busca também é feitas simplesmente chamando a função de busca da árvore AVL

```
int consultaSet(Set* s, int num){
    if(s == NULL)
        return 0;

    return consulta_ArvAVL(s->arv,num);
}
```

Conjunto | Iterando

- Precisamos percorrer todos os elementos do conjunto, como se fosse um array
  - Muito útil quando queremos calcular a união ou a intersecção
- Para realizar essa tarefa, podemos utilizar um *iterator*

```
//percorrendo um array
int i, V[3] = {5,10,15};
for(i = 0; i < 3; i++){
   printf("array: %d:\n",V[i]);
}

//iterator
Set *A = criaSet();
insereSet(A,5);
insereSet(A,10);
insereSet(A,15);
for(beginSet(A); !endSet(A); nextSet(A)){
   getItemSet(A, &x);
   printf("Iterator %d: %d\n",i,x);
}
```

Conjunto | Iterando

- Um *iterator* é um padrão de projeto comportamental
  - Fornece uma forma de acessar, sequencialmente, todos os elementos de uma estrutura de dados, mas sem expor sua representação interna para o usuário

```
//percorrendo um array
int i, V[3] = {5,10,15};
for(i = 0; i < 3; i++){
   printf("array: %d:\n",V[i]);
}

//iterator
Set *A = criaSet();
insereSet(A,5);
insereSet(A,10);
insereSet(A,15);
for(beginSet(A); !endSet(A); nextSet(A)){
   getItemSet(A, &x);
   printf("Iterator %d: %d\n",i,x);
}
```

Conjunto | Iterando

- O *iterator* é um ponteiro para uma lista dinâmica
- Para gerenciar, usamos 4 funções:
  - **beginSet()**: inicializa o iterator
  - **endSet()**: verifica se chegamos ao último elemento da estrutura
  - **nextSet()**: movimenta o *iterator* para o próximo elemento
  - **getItemSet()**: retorna, por referência, o valor do elemento atual no *iterator*

```
//iterator
Set *A = criaSet();
insereSet(A,5);
insereSet(A,10);
insereSet(A,15);
for(beginSet(A); !endSet(A); nextSet(A)){
   getItemSet(A, &x);
   printf("Iterator %d: %d\n",i,x);
}
```

Conjunto | Iterando

- **beginSet()**
  - inicializa o campo *iter* com uma lista dinâmica produzida a partir da árvore AVL
  - função **iterator_ArvAVL()**
- **endSet()**
  - verifica se o campo *iter* é igual a **NULL**
  - final da lista dinâmica encadeada

```
void beginSet(Set *s){
    if(s == NULL)
        return;

    s->iter = NULL;
    iterator_ArvAVL(s->arv, &(s->iter));
}

int endSet(Set *s){
    if(s == NULL)
        return 1;
    if(s->iter == NULL)
        return 1;
    else
        return 0;
}
```

Conjunto | Iterando

- **nextSet()**
  - atribui ao campo *iter* o próximo elemento da lista, se existir.
  - Também libera o elemento anterior
- **getItemSet()**
  - retorna, por referência, o valor associado ao campo *iter*, se este for diferente de **NULL**

```
void nextSet(Set *s){
    if(s == NULL)
        return;
    if(s->iter != NULL){
        struct iterator *no = s->iter;
        s->iter = s->iter->prox;
        free(no);
    }
}

void getItemSet(Set *s, int *num){
    if(s == NULL)
        return;
    if(s->iter != NULL)
        *num = s->iter->valor;
}
```

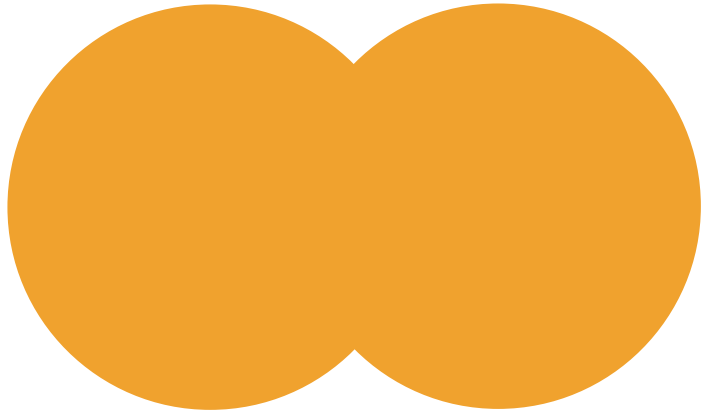
Conjunto | Iterando

- *iter* é só um ponteiro para uma lista dinâmica
  - **struct iterator**
- **iterator_ArvAVL()**
  - Faz o percurso *em-ordem* na AVL e adiciona cada nó da árvore no início da lista dinâmica

```
struct iterator{
    int valor;
    struct iterator *prox;
};

void iterator_ArvAVL(ArvAVL *raiz,struct iterator **iter){
    if(raiz == NULL)
        return;
    if(*raiz != NULL){
        iterator_ArvAVL(&((*raiz)->esq),iter);
        //Insere no início da lista
        struct iterator* no;
        no = (struct iterator*) malloc(sizeof(struct iterator));
        no->valor = (*raiz)->info;
        no->prox = *iter;
        *iter = no;

        iterator_ArvAVL(&((*raiz)->dir),iter);
    }
}
```

Conjunto | União

- Consiste em criar um terceiro conjunto contendo os elementos de ambos os conjuntos, sem repetições
  - A operação de inserção da AVL se encarrega de evitar que um valor repetido seja inserido



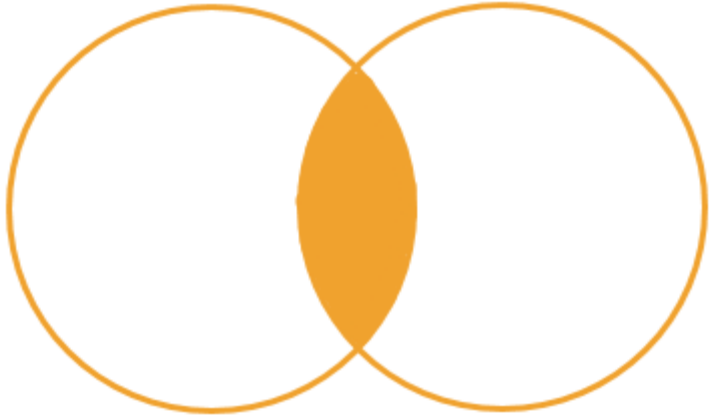
Conjunto | União

- Assim, para fazer a união basta percorrer os dois conjuntos e inserir os elementos deles no novo conjunto
- Para isso, usamos no *iterator* definido anteriormente

```
Set* uniaoSet(Set* A, Set* B){
    if(A == NULL || B == NULL)
        return NULL;
    int x;
    Set *C = criaSet();

    for(beginSet(A); !endSet(A); nextSet(A)){
        getItemSet(A, &x);
        insereSet(C,x);
    }

    for(beginSet(B); !endSet(B); nextSet(B)){
        getItemSet(B, &x);
        insereSet(C,x);
    }

    return C;
}
```

Conjunto | Intersecção

- Consiste em criar um terceiro conjunto contendo os elementos que existem em ambos os conjuntos
  - Pode ser facilmente feito percorrendo o menor dos conjuntos e verificando se cada um dos seus elementos existe no conjunto maior



Conjunto | Intersecção

- A intersecção terá no máximo a mesma quantidade de elementos do conjunto menor
  - Por questão de eficiência, é mais rápido percorrer o conjunto menor e verificar se seus elementos existem no conjunto maior

```
Set* interseccaoSet(Set* A, Set* B){
    if(A == NULL || B == NULL)
        return NULL;
    int x;
    Set *C = criaSet();

    if(tamanhoSet(A) < tamanhoSet(B)){
        for(beginSet(A); !endSet(A); nextSet(A)){
            getItemSet(A, &x);
            if(consultaSet(B,x))
                insereSet(C,x);
        }
    }else{
        for(beginSet(B); !endSet(B); nextSet(B)){
            getItemSet(B, &x);
            if(consultaSet(A,x))
                insereSet(C,x);
        }
    }
    return C;
}
```

MULTICONJUNTO

Multiconjunto | Definição

- O multiconjunto (*Bag*) é um conjunto não ordenado de elementos, sendo permitida a repetição de elementos
  - Se os elementos forem do mesmo tipo, temos um multiconjunto homogêneo.

Multiconjunto | Definição

- É uma estrutura similar a um carrinho de compras
  - Você pode adicionar uma ou mais ocorrências de um item, sem se importar com a ordem
- Pode ser implementado usando
  - Array estático
  - Lista dinâmica encadeada

Multiconjunto | Aplicações

- Contagem de frequências em processamento de texto
  - Um multiconjunto pode armazenar as palavras como chaves e suas frequências como valores
- Modelagem de inventário em jogos ou comércio
  - Em sistemas de inventário onde vários itens do mesmo tipo podem existir (como em jogos ou sistemas de estoque), multiconjuntos podem armazenar o número de ocorrências de cada item
- Contagem de eventos em sistemas de monitoramento
  - Sistemas que monitoram eventos (como logs de erros em servidores) podem usar multiconjuntos para armazenar os tipos de eventos e suas frequências

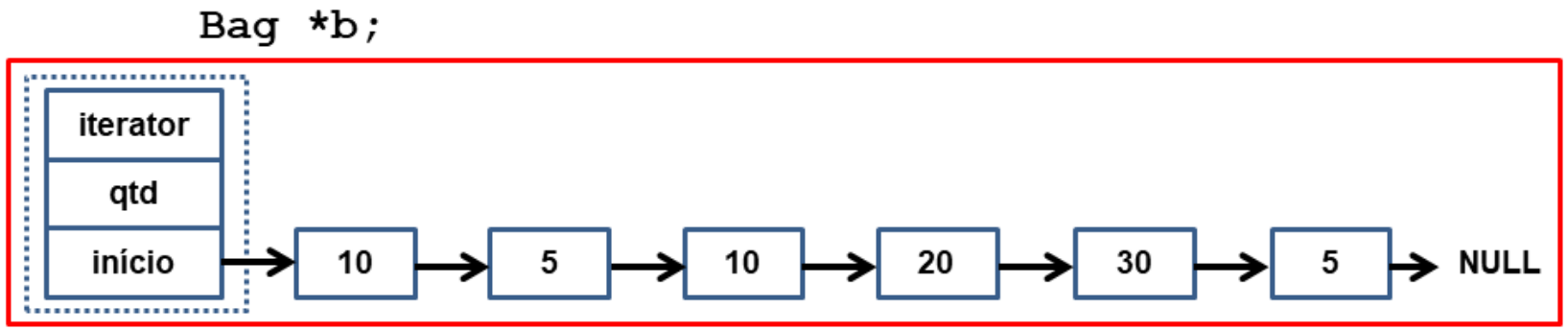
Multiconjunto | TAD

- Utiliza uma lista dinâmica para armazenar os elementos
  - Custo máximo da inserção: *O(1)*
  - Busca e remoção: *O(N)*
- O *iterator* é outra lista dinâmica
  - Será definido mais tarde seu funcionamento

```
//Definição do tipo Bag
struct NO{
    int valor;
    struct NO *prox;
};

//Definição do Nó Descritor
struct descritor{
    struct NO *inicio;
    int qtd;
    struct NO *iterator;
};

typedef struct descritor Bag;
```

Multiconjunto | TAD

- O multiconjunto é mantido por um nó descritor contendo três campos
  - início
  - qtd
  - um ponteiro para criar um *iterator*



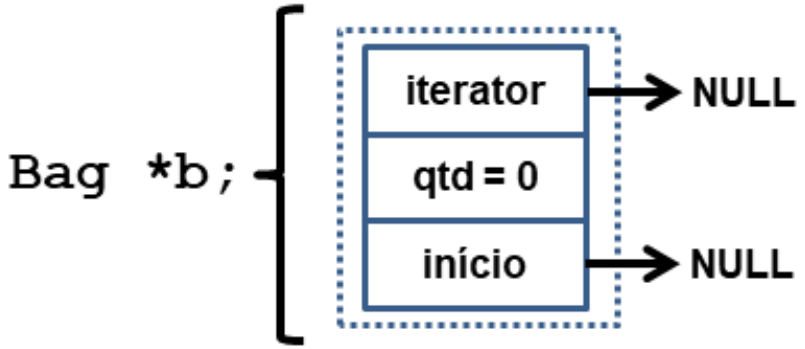
Multiconjunto | Criando e destruindo

- Criação
  - Aloca uma área de memória para o multiconjunto
  - Corresponde a memória necessária para armazenar a estrutura do multiconjunto
  - Inicializa os demais campos

```
Bag* criaBag(){
    Bag* b = (Bag*) malloc(sizeof(Bag));
    if(b != NULL){
        b->inicio = NULL;
        b->qtd = 0;
        b->iterator = NULL;
    }
    return b;
}
```

Multiconjunto | Criando e destruindo

- Ao final desse processo temos um multiconjunto vazio
- Podemos verificar se o multiconjunto está vazio pelos campos
  - qtd é igual a 0?
  - início é igual a NULL?



Multiconjunto | Criando e destruindo

- Liberação é feita em três etapas
  - Liberar a memória alocada para a lista dinâmica
  - Liberar a memória alocada para a estrutura multiconjunto

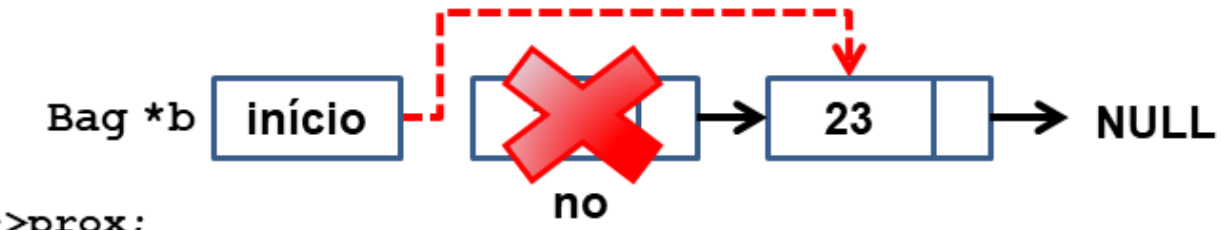
```
void liberaBag(Bag* b){
    if(b != NULL){
        struct NO* no;
        while(b->inicio != NULL){
            no = b->inicio;
            b->inicio = b->inicio->prox;
            free(no);
        }
        free(b);
    }
}
```

Multiconjunto | Criando e destruindo

Multiconjunto inicial

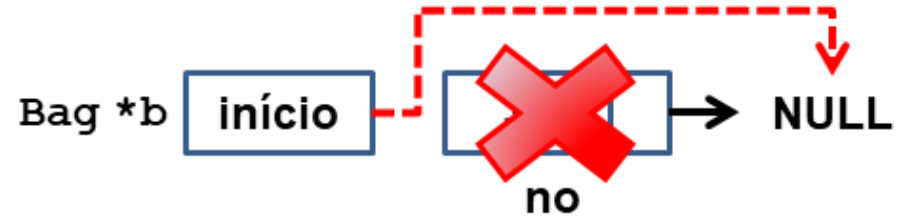
Passo 1:

```
no = b->inicio;
b->inicio= b->inicio->prox;
free(no);
```



Passo 2:

```
no = b->inicio;
b->inicio= b->inicio->prox;
free(no);
```



Fim:

```
B->inicio == NULL
```

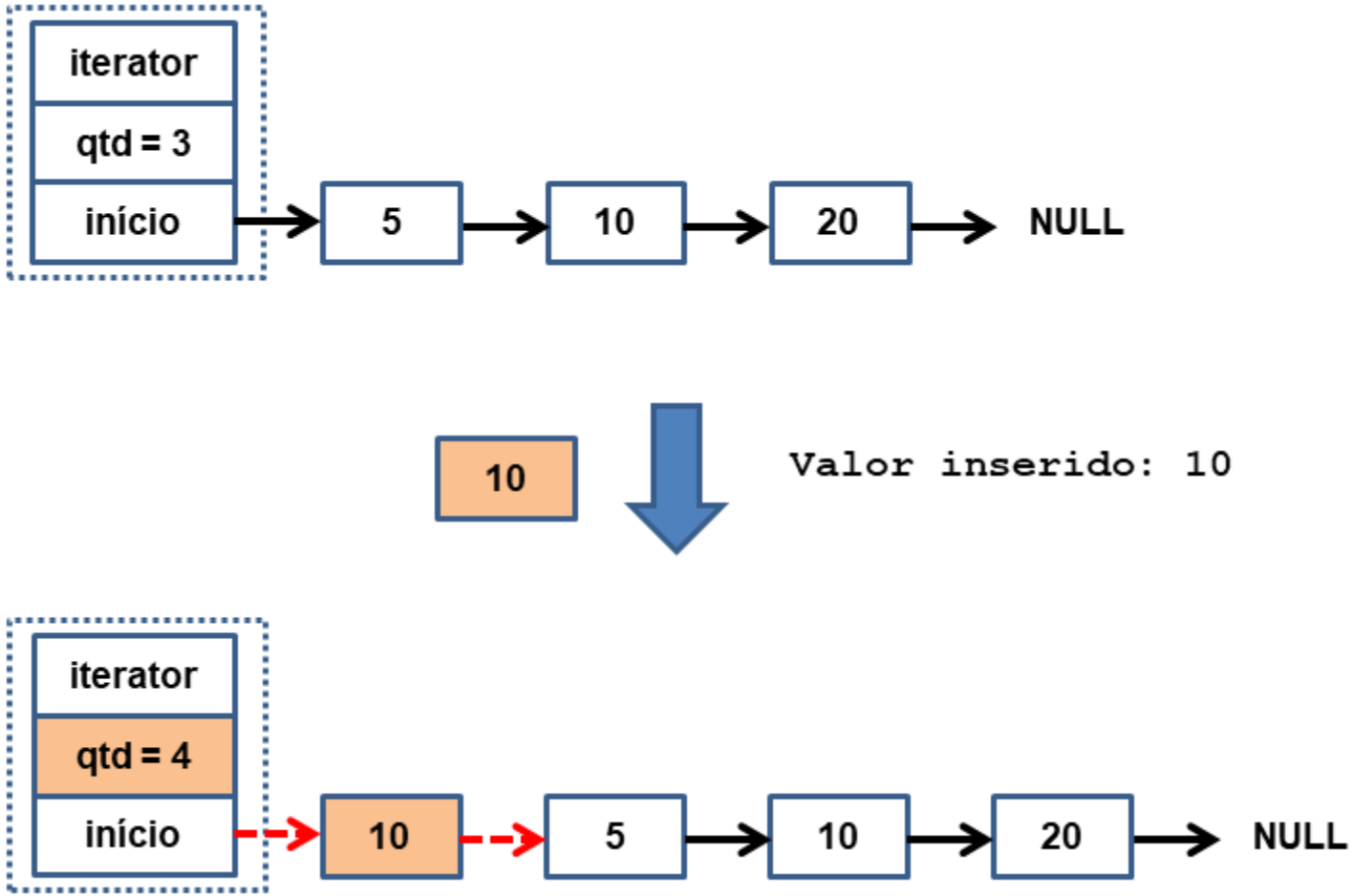
Multiconjunto | Inserção

- Tarefa simples, envolve alocar espaço para o novo elemento e ajustar alguns ponteiros

- Primeiro, verificamos se o multiconjunto existe
- Em seguida
  - alocar memória para o novo nó
  - copiar os dados
  - mudar o início

```
int insereBag(Bag* b, int num){
    if(b == NULL)
        return 0;
    struct NO* no;
    no = (struct NO*) malloc(sizeof(struct NO));
    if(no == NULL)
        return 0;
    no->valor = num;
    no->prox = b->inicio;
    b->inicio = no;
    b->qtd++;
    return 1;
}
```

Multiconjunto | Inserção



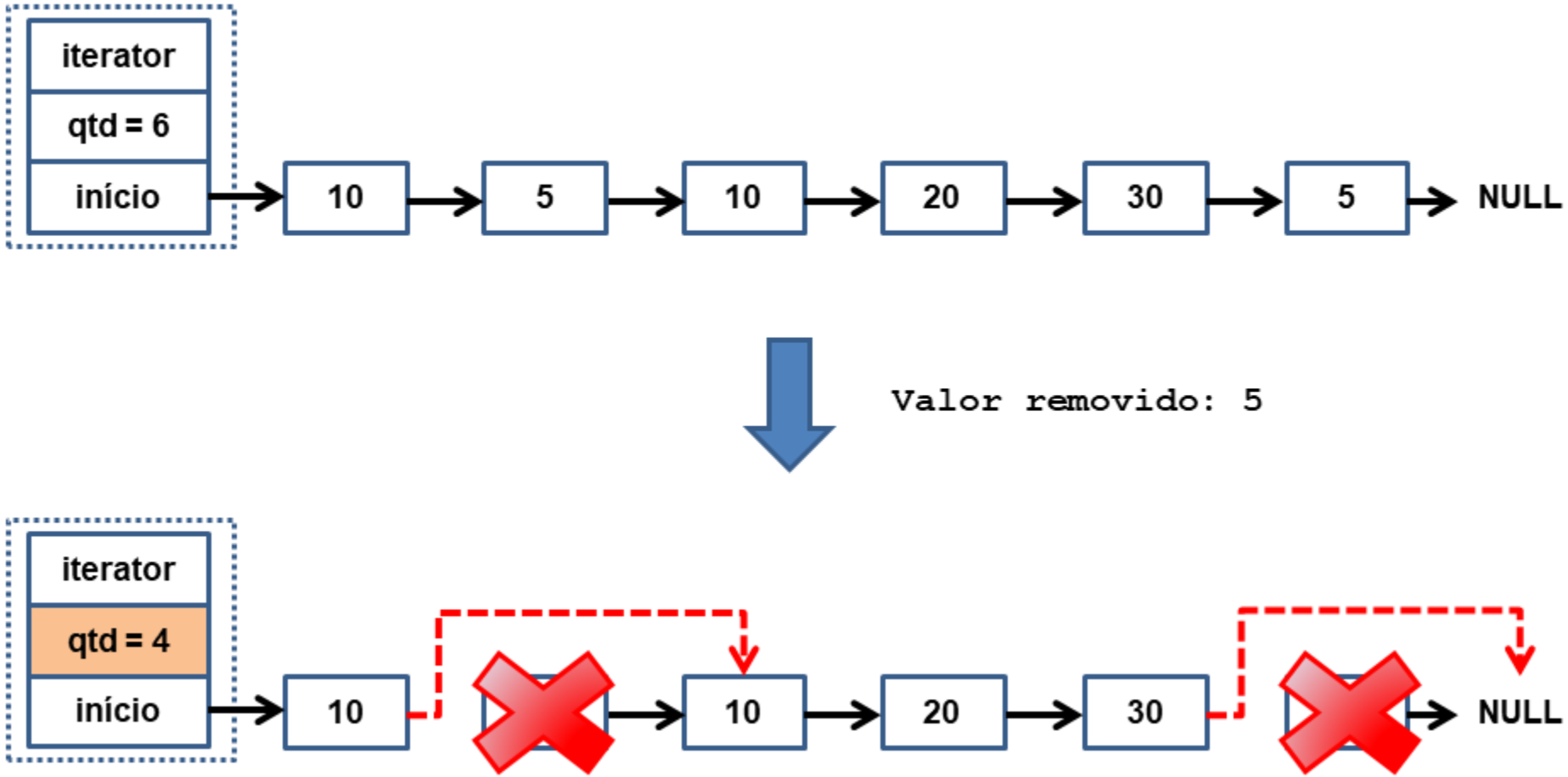
Multiconjunto | Remoção

- A remoção deve considerar que o multiconjunto permite repetição

- Envolve percorre toda a lista e
  - liberar a memória dos nós com o elemento buscado
  - ajustar alguns ponteiros e, em alguns caso, o inicio da lista

```
int removeBag(Bag* b, int num){
    if(b == NULL)  return 0;
    int cont = 0;
    struct NO *ant;
    struct NO *no = b->inicio;
    while(no != NULL){
        if(no->valor == num){
            cont++;
            if(b->inicio == no){//remover o primeiro?
                b->inicio = no->prox;
                free(no);
                no = b->inicio;
            }else{
                ant->prox = no->prox;
                free(no);
                no = ant->prox;
            }
        }else{
            ant = no;
            no = no->prox;
        }
    }
    b->qtd = b->qtd - cont;
    return cont;
}
```

Multiconjunto | Remoção



Multiconjunto | Busca

- Basicamente, envolve percorrer a lista dinâmica que define o multiconjunto
- Optamos por retornar apenas a quantidade de elementos com o valor procurado

```
int consultaBag(Bag* b, int num){
    if(b == NULL)
        return 0;
    struct NO *no = b->inicio;
    int cont = 0;
    while(no != NULL){
        if(no->valor == num)
            cont++;
        no = no->prox;
    }

    return cont;
}
```

Multiconjunto | Iterando

- Precisamos percorrer todos os elementos do multiconjunto, como se fosse um array
  - Muito útil quando queremos calcular a união ou a intersecção
- Para realizar essa tarefa, podemos utilizar um *iterator*

```
//percorrendo um array
int i, V[3] = {5,10,15};
for(i = 0; i < 3; i++){
    printf("array: %d:\n",V[i]);
}

    //iterator
Bag *b = criaBag();
insereBag(b,5);
insereBag(b,10);
insereBag(b,15);

for(beginBag(b); !endBag(b); nextBag(b)){
    getItemBag(b, &x);
    printf("Iterator %d:\n",x);
}
```

Multiconjunto | Iterando

- Um *iterator* é um padrão de projeto comportamental
  - Fornece uma forma de acessar, sequencialmente, todos os elementos de uma estrutura de dados, mas sem expor sua representação interna para o usuário

```
//percorrendo um array
int i, V[3] = {5,10,15};
for(i = 0; i < 3; i++){
    printf("array: %d:\n",V[i]);
}

    //iterator
Bag *b = criaBag();
insereBag(b,5);
insereBag(b,10);
insereBag(b,15);

for(beginBag(b); !endBag(b); nextBag(b)){
    getItemBag(b, &x);
    printf("Iterator %d:\n",x);
}
```

Multiconjunto | Iterando

- O *iterator* é um ponteiro para uma lista dinâmica
- Para gerenciar, usamos 4 funções:
  - **beginSet()**: inicializa o iterator
  - **endSet()**: verifica se chegamos ao último elemento da estrutura
  - **nextSet()**: movimenta o *iterator* para o próximo elemento
  - **getItemSet()**: retorna, por referência, o valor do elemento atual no *iterator*

```
    //iterator
Bag *b = criaBag();
insereBag(b,5);
insereBag(b,10);
insereBag(b,15);

for(beginBag(b); !endBag(b); nextBag(b)){
    getItemBag(b, &x);
    printf("Iterator %d:\n",x);
}
```

Multiconjunto | Iterando

- **beginSet()**
  - inicializa o campo *iter* com o campo inicio do multiconjunto
- **endSet()**
  - verifica se o campo *iter* é igual a **NULL**
  - final da lista dinâmica encadeada

```
void beginBag(Bag *b){
    if(b == NULL)
        return;
    b->iterator = b->inicio;
}

int endBag(Bag *b){
    if(b == NULL)
        return 1;
    if(b->iterator == NULL)
        return 1;
    else
        return 0;
}
```

Multiconjunto | Iterando

- **nextSet()**
  - atribui ao campo iterator o próximo elemento da lista, se ele existir
- **getItemSet()**
  - retorna, por referência, o valor associado ao campo *iter*, se este for diferente de **NULL**

```
void nextBag(Bag *b){
    if(b == NULL)
        return;
    if(b->iterator != NULL)
        b->iterator = b->iterator->prox;
}

void getItemBag(Bag *b, int *num){
    if(b == NULL)
        return;
    if(b->iterator != NULL)
        *num = b->iterator->valor;
}
```

Material Complementar | Vídeo Aulas

- Aula 131 - Conjunto: definição
  - https://youtu.be/Mvdiel_QqKY
- Aula 132 - Conjunto: implementação
  - https://youtu.be/JAqXM9IwBcU
- Aula 133 - Conjunto: inserção, remoção e busca
  - https://youtu.be/bdB39oiHTSI
- Aula 134 - Conjunto: criando um iterator
  - https://youtu.be/lpR5UZy7EsY
- Aula 135 - Conjunto: união e intersecção
  - https://youtu.be/9wHaJNpENUo

Material Complementar | Vídeo Aulas

- Aula 136 – Multiconjunto (Bag): definição
  - https://youtu.be/-TLuRW222U0
- Aula 137 - Multiconjunto (Bag): implementação
  - https://youtu.be/0C9m7SSqE9w
- Aula 138 - Multiconjunto (Bag): inserção, remoção e busca
  - https://youtu.be/29kcHjh3Fn4
- Aula 139 - Multiconjunto (Bag): criando um iterator
  - https://youtu.be/IhA4P-iahR4

Material Complementar | GitHub

- https://github.com/arbackes