Estruturas de Dados
Resolução de exercícios de testes

Eduardo Freitas Fernandes

2025

Exercício 1

```c
int hash(char matricula[6]) {
    unsigned int hash = 0;
    unsigned int base = 31; // número primo

    while (*matricula != '\0') {
        hash = hash * base + (unsigned int) (*matricula);
        matricula++;
    }

    return hash;
}
```

```c
int insert(Tabela t, char matricula[6]) {
    int index = hash(matricula) % SIZE;
    struct no ** temp = &(t[index]);

    while (*temp != NULL && strcmp((*temp)->matricula, matricula))
        temp = &((*temp)->next);

    if (*temp == NULL) {
        *temp = malloc(sizeof(struct no));
        strcpy((*temp)->matricula, matricula);
        (*temp)->next = NULL;
    }

    return index;
}
```

Exercício 2

```c
#define LEFT (-1)
#define BALANCED 0
#define RIGHT 1

Arvore rr(Arvore arv) {
    Arvore left = arv->esq;
    int root_status = arv->bal;
    int left_status = left->bal;

    arv->esq = left->dir;
    left->dir = arv;

    if (root_status == RIGHT && left_status <= BALANCED)
        arv->bal = BALANCED;
    else
        arv->bal = LEFT;

    return left;
}
```

Exercício 3

$$T(N) = \begin{cases} 0 & N \leq 0 \\ 2 \times T_{altura}(N/2) + 2 \times T(N/2) & N > 0 \end{cases}$$

$$= \begin{cases} 0 & N \leq 0 \\ N + 2 \times T(N/2) & N > 0 \end{cases} = \sum_{i=1}^{1+\log_2(N)} N = \Theta(N \times \log_2(N))$$

Exercício 4

```c
void sinonimos(Dic d, char *pal) {
    int index = hash(pal) % TAM;
    struct p *temp = d[index];

    while (temp != NULL && strcmp(temp->pal, pal))
        temp = temp->next;

    if (temp != NULL) {
        struct s *aux = temp->sins;
        while (aux != NULL) {
            printf("'%s' == '%s'\n", pal, aux->sin);
            aux = aux->next;
        }
    }
}
```

Exercício 5

```
Tree heapToTreeRec(Heap h, int i, int size) {
    if (i >= size)
        return NULL;

    Tree root = malloc(sizeof(struct nodo));
    root->val = h[i];
    root->esq = heapToTreeRec(h, 2 * i + 1, size);
    root->dir = heapToTreeRec(h, 2 * i + 2, size);

    return root;
}

Tree heapToTree(Heap h, int size) {
    return heapToTreeRec(h, 0, size);
}
```

Exercício 6

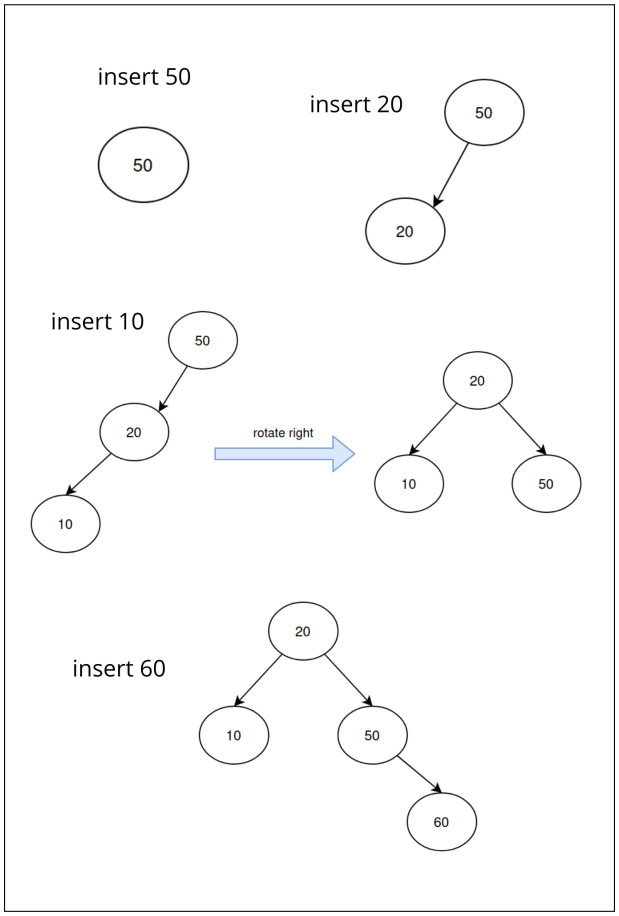

Figura 1: Inserir 50, 20, 10, 60

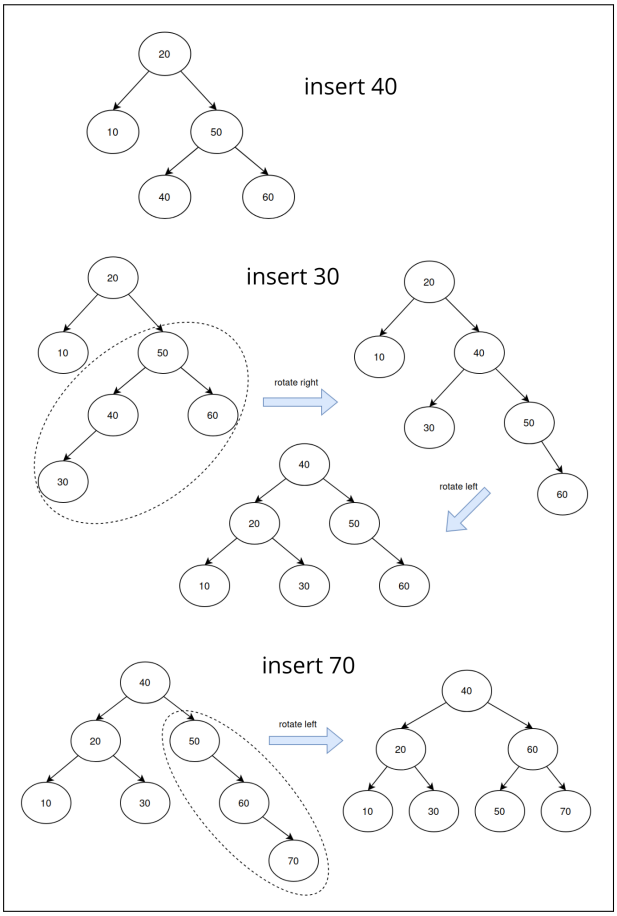

Figura 2: Inserir 40, 30, 70

Exercício 7

```
Node buildBST(int arr[], int n) {
    if (n <= 0) return NULL;

    int middle = n / 2;
    Node root = malloc(sizeof(struct node));
    root->info = arr[middle];
    root->esq = buildBST(arr, middle);
    root->dir = buildBST(arr + middle, n - middle);

    return root;
}
```

Exercício 8

```
#define SIZE 1001

typedef struct count {
    char store[SIZE];
    int total;
} Count;

void init_count(Count *c) {
    c->total = 0;

    for (int i = 0; i < SIZE; i++)
        c->store[i] = 0;
}

void insert(Count *c, int x) {
    c->total += c->store[x] == 0 ? 1 : 0;
    c->store[x] = 1;
}

int check(Count *c, int x) {
    return c->store[x];
}

void show(Count *c) {
    int count = 0, i;
    for (i = 0; i < SIZE && count < c->total; i++) {
        if (c->store[i] != 0)
            printf("%d ", i);
        count += c->store[i];
    }
}
```

Exercício 9

```c
#define PARENT(i) ((i - 1) / 2)
#define LEFT(i) (2 * i + 1)
#define RIGHT(i) (2 * i + 2)

void swap(MinHeap h, int i, int j) {
    int temp = h->heap[i];
    h->heap[i] = h->heap[j];
    h->heap[j] = temp;
}

void bubbleUp(MinHeap h, int i) {
    int parent = PARENT(i);
    while (i > 0 && h->heap[parent] > h->heap[i]) {
        swap(h, i, parent);
        i = parent;
        parent = PARENT(parent);
    }
}

void muda (MinHeap h, int pos, int valor) {
    h->heap[pos] = valor;
    bubbleUp(h, pos);
}
```

Exercício 10

```c
int procura (ABPInt a, int l, int u) {
    while (a != NULL) {
        if (a->valor < u && a->valor > l)
            return 1;
        if (a->valor >= u)
            a = a->esq;
        else if (a->valor <= l)
            a = a->dir;
    }
    return 0;
}
```

Exercício 11

Análise de complexidade da função `procura()`:

- **Melhor Caso**: a raiz da árvore está entre `l` e `u`.
- **Pior Caso**: os valores guardados na árvore não estão presentes no intervalo definido por `l` e `u`, ou estão no último nível da árvore.

$$T(N) = \sum_{i=1}^{1+\log_2(N)} 1 = 1 + \log_2(N) = \Theta(\log_2(N))$$

Exercício 12

```c
int biggest(MinHeap *h) {
    // as folhas da heap são os maiores elementos
    // logo basta percorrer apenas esses elementos
    int stop = h->tamanho / 2;
    int max = h->heap[h->tamanho - 1];
    for (int i = h->tamanho - 2; i >= 0 && i >= stop; i--)
        if (h->heap[i] > max)
            max = h->heap[i];

    return max;
}
```

Exercício 13

```c
int rdir(AVL *tptr) {
    if (*tptr == NULL || (*tptr)->esq == NULL)
        return -1;

    AVL left = (*tptr)->esq;
    (*tptr)->esq = left->dir;
    *tptr = left;
    return 0;
}
```

Exercício 14

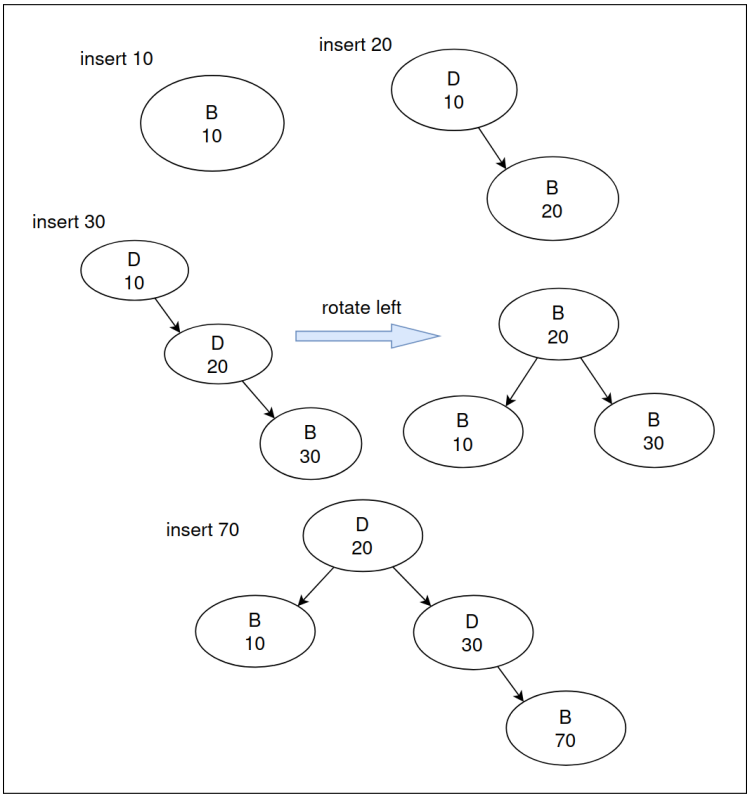

Figura 3: Inserir 10, 20, 30, 70

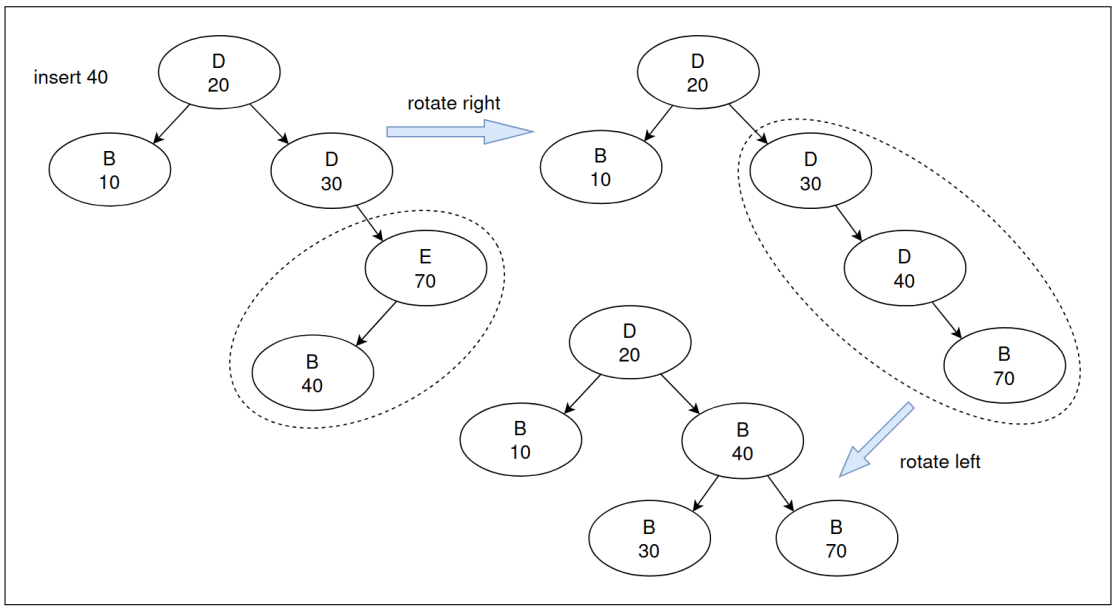

Figura 4: Inserir 40

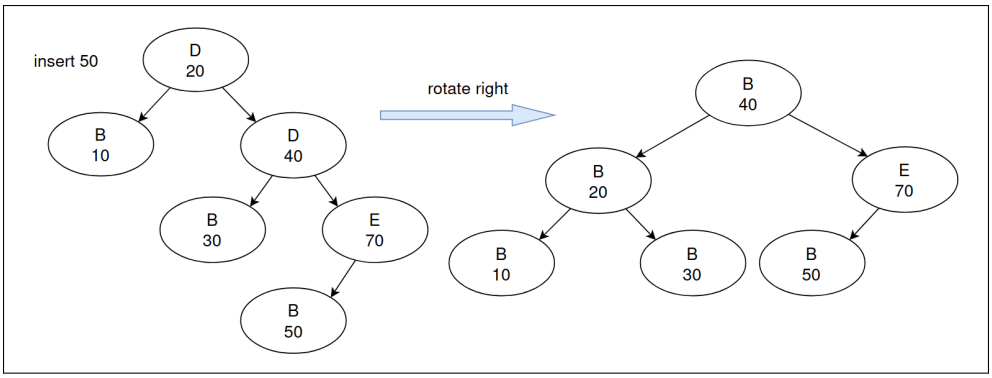

Figura 5: Inserir 50

Exercício 15

Tabelas de Hash:

a) quando todas as `n` inserções colidem na mesma posição, tornando a tabela de hash numa lista ligada, com complexidade de O(N) na pesquisa.

b) sendo que não existe uma relação entre os elementos da tabela, para percorrê-los de forma ordenada, seria necessário recolhê-los para uma estrutura de dados, um array por exemplo. Esta operação teria um custo linear, $\Theta(N)$. De seguida seria necessário ordenar o array, suponhamos que o custo seria $\Theta(N \times \log(N))$. Logo o tempo de execução seria $\Theta(N \times \log(N))$.

c) as Hash Tables são estruturas de dados eficientes em consultas, se a função de hash for boa e distribuir bem os elementos, em média teremos uma complexidade de $\Theta(1)$. Mas se for necessário ter monotonia entre os elementos, as Hash Tables são pouco eficientes, pois não estabelecem ordem entre os elementos.

Exercício 16

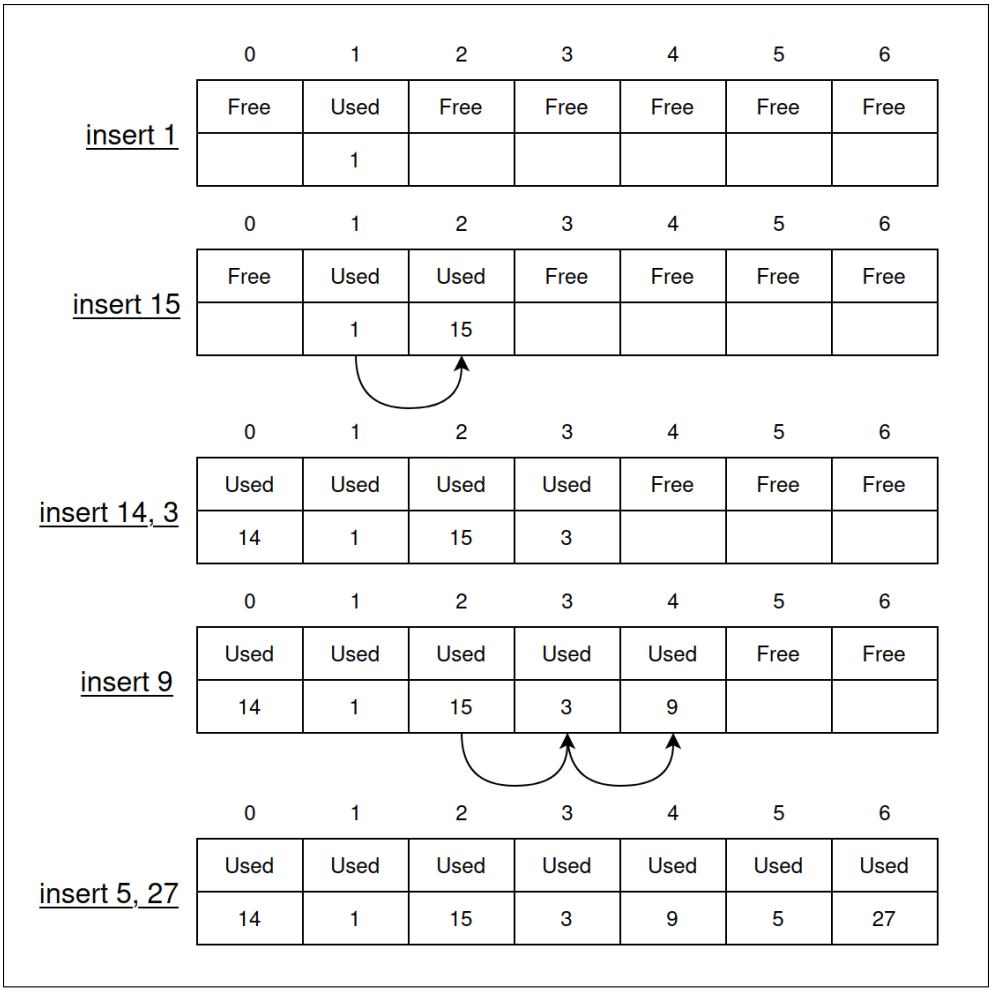

Figura 6: Inserir 1, 15, 14, 3, 9, 5 e 27

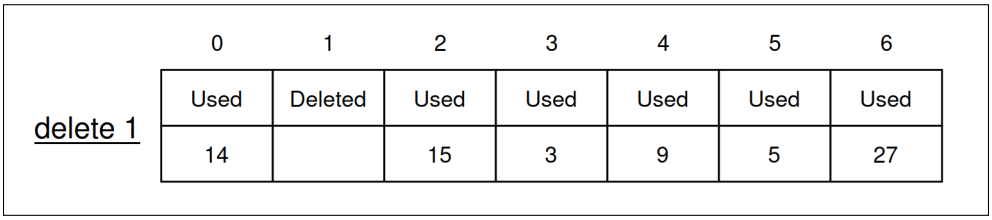

Figura 7: Remover 1

Exercício 17

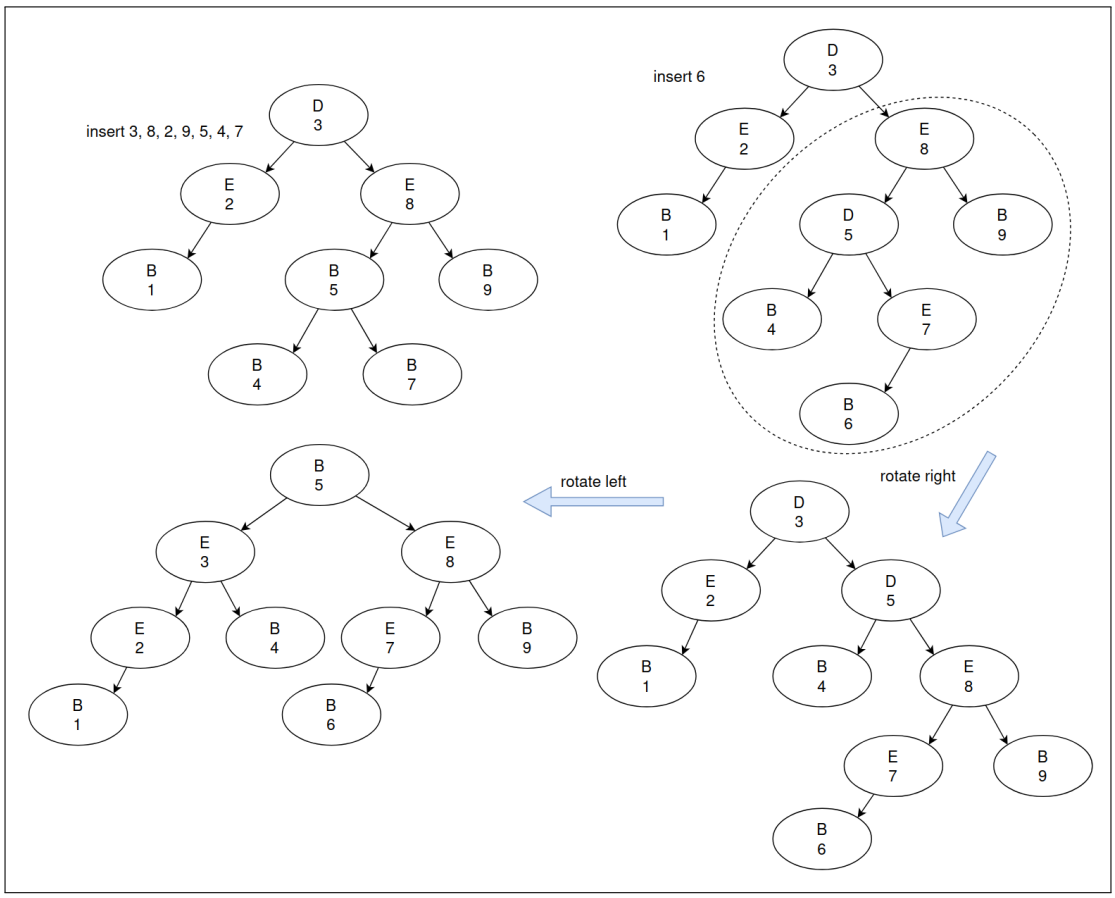

Figura 8: Árvore AVL

Exercício 18

Tabela de Hash com resolução de colisões por Chaining:

- o fator de carga em tabelas de Hash com resolução de colisões por Chaining corresponde ao tamanho médio de cada lista da tabela, ou seja, o número médio de colisões. O valor máximo possível é uma lista ter comprimento N.
- se o fator de carga está a 50%, no melhor caso, metade das listas da tabela terão apenas um elemento, e numa procura falhada, será feita no máximo uma comparação, que dará falso. Logo no melhor caso a complexidade será de $\Theta(1)$. No pior caso, uma das listas terá $\frac{N}{2}$ elementos (N corresponde ao tamanho da tabela), enquanto que as restantes estarão vazias. Numa procura falhada será necessário percorrer essa lista até ao fim, logo a complexidade no pior caso é de $\Theta(N)$.

Exercício 19

```c
void swap(int * array, int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

int *ordenados (MinHeap h) {
    int *ord = calloc(h->used, sizeof(int));

    for (int i = 0; i < h->used; i++) {
        ord[i] = h->values[0];
        swap(h->values, 0, h->used - 1 - i);
        bubbleDown(h); // coloca a raiz no nível correto
    }

    free(h->values);
    free(h);

    return ord;
}
```

Exercício 20

Nesta tabela, o fator de carga, é o tamanho médio de cada lista, ou seja, o número médio de colisões, logo: `load = used / size`

```c
float loadFactorC (HashTableChain t) {
    int count = 0;
    EntryChain *temp = NULL;
    for (int i = 0; i < t->hashsize; i++) {
        temp = (t->table) + i;
        while(temp != NULL) {
            count++;
            temp = temp->next;
        }
    }

    return count / (float) t->hashsize;
}
```

Exercício 21

```c
int hash (int size, char key[]) {
    int res = 0;
    for (int i = 0; key[i] != '\0'; i++)
        res += rank[tolower(key[i]) - 'a'];

    return res % size;
}
```

Exercício 22

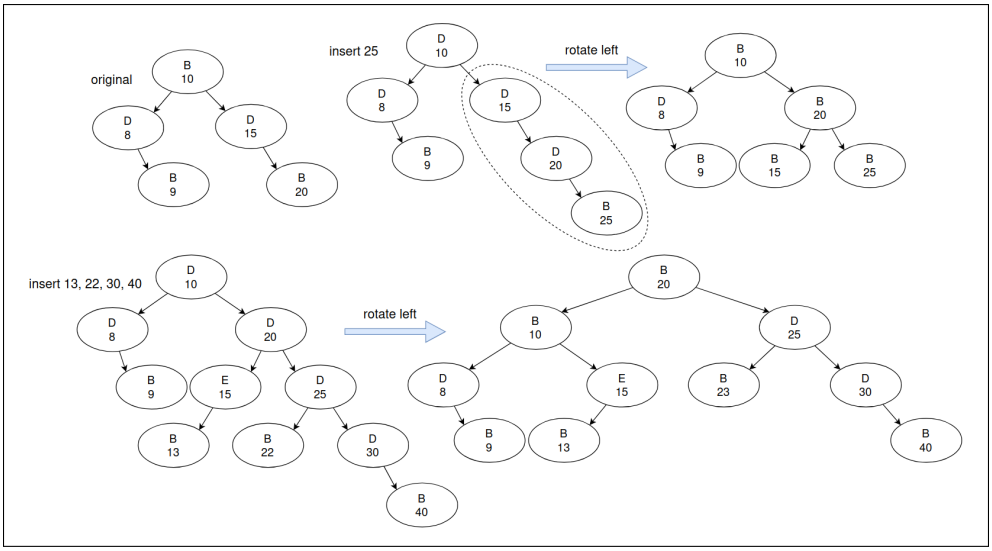

Figura 9: Inserir 25, 13, 22, 30, 40

Exercício 23

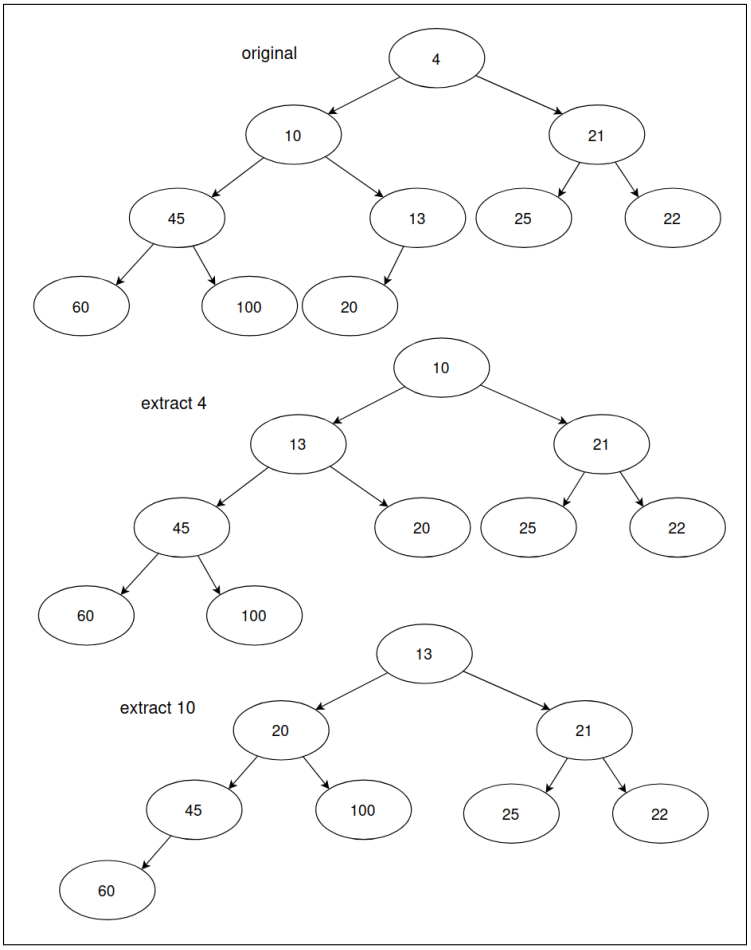

Figura 10: Remover 4, 10

$$T_{hToAux}(N) = \begin{cases} 0 & N \leq 0 \\ 1 + T_{hToAux}(N/2) & N > 0 \end{cases}$$

$$= \sum_{i=0}^{1+\log_2(N)} 2^i = 4 \times N - 1 = \Theta(N)$$

Exercício 24

```c
int hash(char * key);

void doubleTable (HashTableOAddr h) {
    int d_size = 2 * h->size, index = 0;
    EntryOAdd * new = calloc(d_size, sizeof(EntryOAdd));
    // não é necessário inicializar os buckets a FREE
    // calloc() faz isso

    for (int i = 0; i < h->size; i++) {
        index = hash(h->table[i].key) % d_size;
        while(new[index].status != STATUS_FREE)
            index = (index + 1) % d_size;

        new[index].status = STATUS_USED;
        new[index].info = h->table[i].info;
        new[index].key = h->table[i].key;
    }

    h->size = d_size;
    free(h->table);
    h->table = new;
}
```

Exercício 25

```c
#define SIZE 1001
typedef struct count {
    int store[SIZE];
    int total;
} Count;

void init_count(Count *c) {
    c->total = 0;
    for (int i = 0; i < SIZE; i++)
        c->store[i] = 0;
}

void insert(Count *c, int x) {
    c->store[x]++;
    c->total++;
}
```

```c
int check(Count *c, int x) { return c->store[x]; }

void show(Count *c) {
    int count = 0, i, j;
    for (i = 0; i < SIZE && count < c->total; i++) {
        for (j = 0; j < c->store[i]; j++)
            printf("%d ", i);
        count += c->store[i];
    }
}
```

Exercício 26

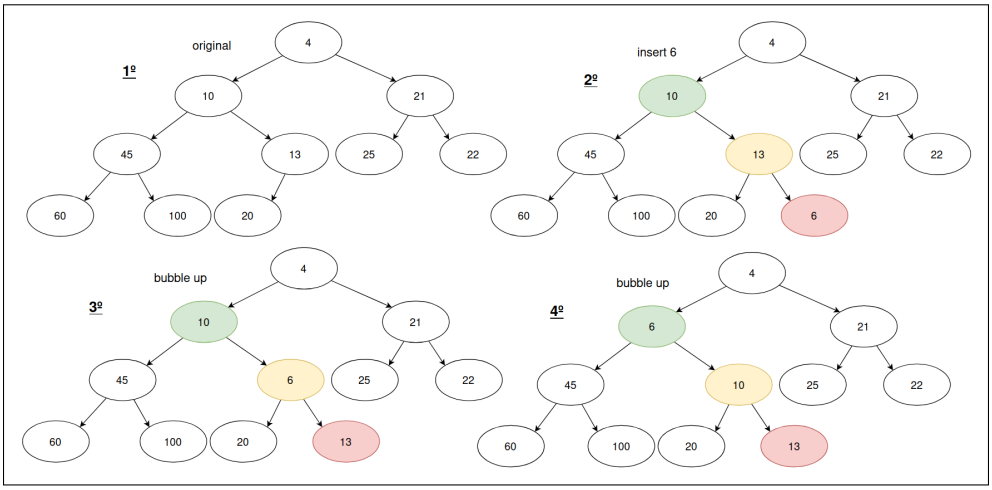

Figura 11: Inserir 6

```c
#define PARENT(i) ((i - 1) / 2)
#define LEFT(i) (2 * i + 1)
#define RIGHT(i) (2 * i + 2)

int menor(MinHeap h, int i) {
    if (i >= h->used) return 1;

    int left = LEFT(i);
    if (left < h->used && h->values[i] > h->values[left])
        return 0;

    int right = RIGHT(i);
    if (right < h->used && h->values[i] > h->values[right])
        return 0;

    return menor(h, left) && menor(h, right);
}

int minHeapOk(MinHeap h) { return menor(h, 0); }
```

Exercício 27

```
HashTable newTable(int hashsize) {
    HashTable new = malloc(sizeof(struct hashT));
    new->hashsize = hashsize;
    new->table = malloc(hashsize * sizeof(Entry));
    for (int i = 0; i < hashsize; i++)
        new->table[i] = NULL;

    return new;
}
```

Exercício 28

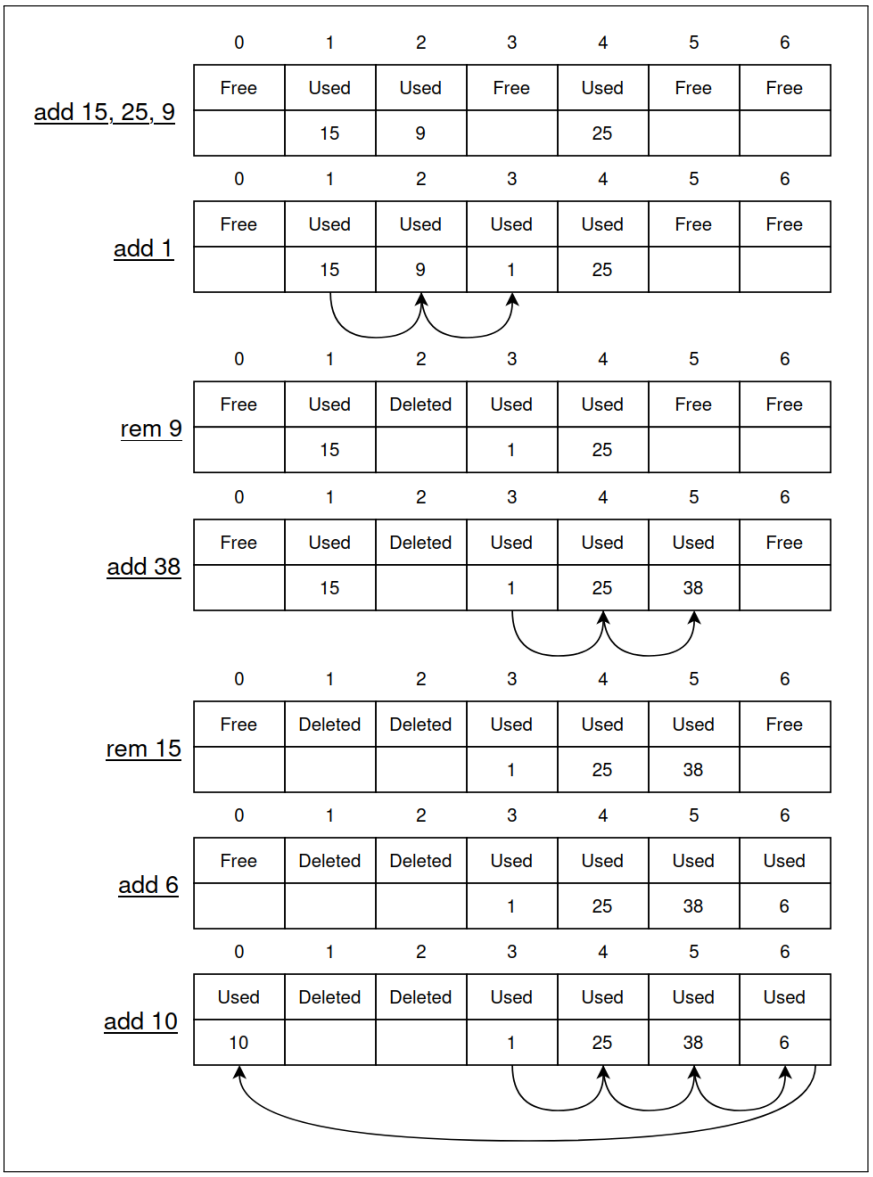

Figura 12: Exercício 28

Exercício 29

```
int altura(AVL a) {
    int height = 0;
    while (a != NULL) {
        a = a->bal == Dir ? a->dir : a->esq;
        height++;
    }

    return height;
}
```

Exercício 30

```
AVL rotateLeft(AVL a) {
    AVL right = a->dir;
    int root_height = a->altura;
    int left_diff = root_height - a->esq->altura;
    int right_diff = root_height - right->altura;

    if (left_diff <= right_diff) {
        right->altura = root_height + 1;
    } else {
        a->altura--;
        right->altura++;
    }

    a->dir = right->esq;
    right->esq = a;

    return right;
}
```

Exercício 31

```
void remApagados(THash h) {
    int i, index;
    struct celula * temp = malloc(sizeof(struct celula) * h->tamanho);
    for (i = 0; i < h->tamanho; i++)
        h->Tabela[i].estado = Livre;

    for (i = 0; i < h->tamanho; i++) {
        if (h->Tabela[i].estado == Ocupado) {
            index = hash(h->Tabela[i].k, h->tamanho);
            while (temp[index].estado == Ocupado)
                index = (index + 1) % h->tamanho;

            temp[index].estado = Ocupado;
            temp[index].info = h->Tabela[i].info;
            temp[index].k = h->Tabela[i].k;
```

```
        }
    }

    h->apagados = 0;
    free(h->Tabela);
    h->Tabela = temp;
}
```

Exercício 32

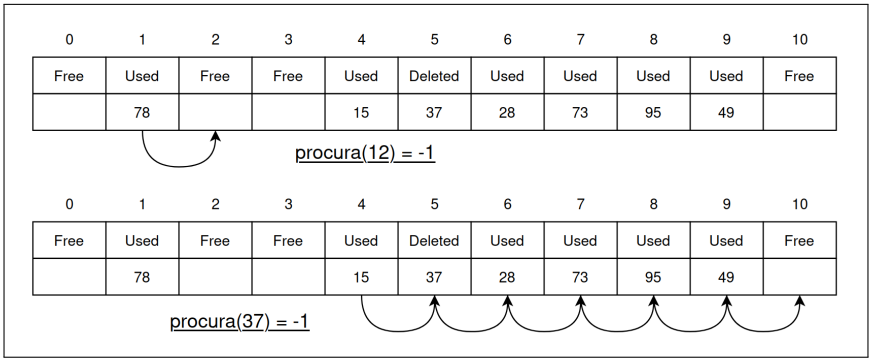

Figura 13: Procurar 12 e 37

Neste exercício estou a supor que se pode inserir valores duplicados, daí a procura de 37 continuar após o indíce 5, caso contrário a procura terminava na posição 5.

Exercício 33

O valor de balanço presente em cada nodo da árvore indica o lado da árvore que tem maior altura, logo seguindo o caminho indicado pelo balanço dos nodos, chegamos ao nível mais profundo da árvore, obtendo a sua altura. A complexidade é logarítmica, pois a função percorre todos os níveis da árvore, apenas uma vez.

```
int altura(AVL arv) {
    int height = 0;

    while (arv != NULL) {
        arv = arv->balanco == Dir ? arv->dir : arv->esq;
        height++;
    }

    return height;
}
```

Exercício 34

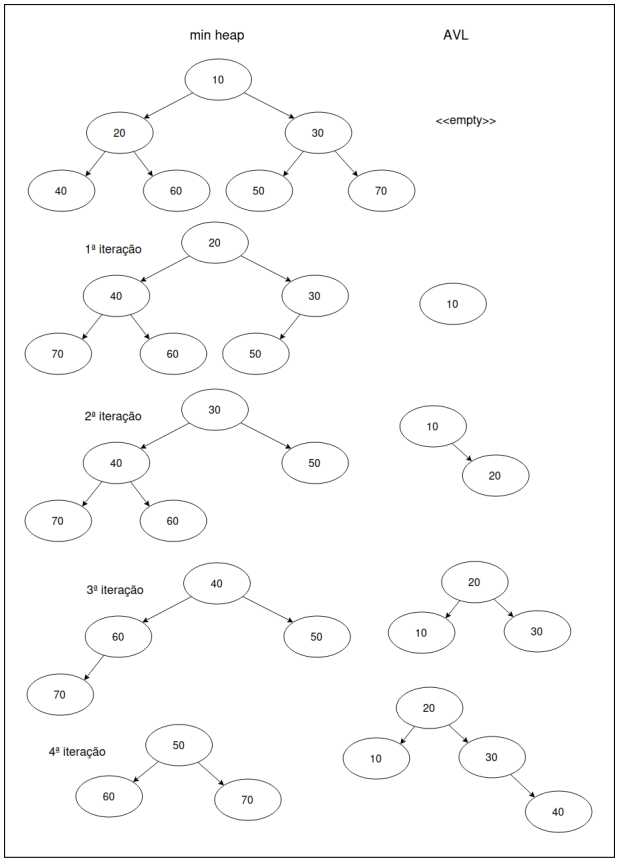

Figura 14: Iterações 1 a 4

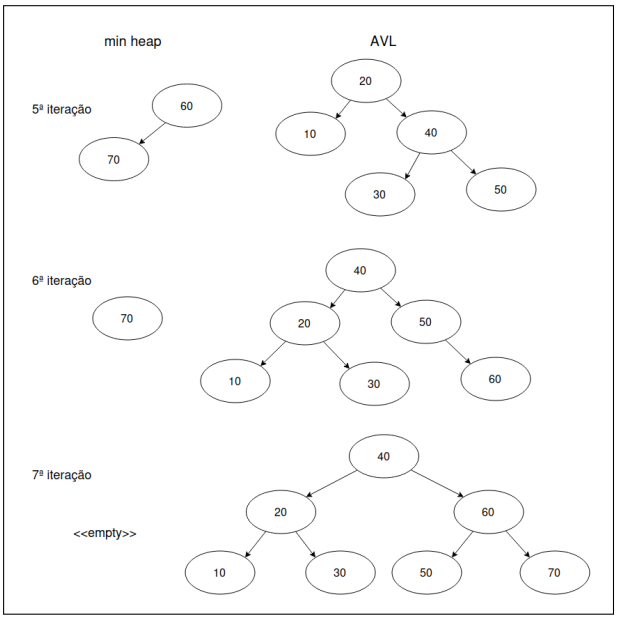

Figura 15: Iterações 5 a 7

Ambas as funções `extractMin()` e `insertTree()` têm uma complexidade logarítmica. Nas primeiras iterações do ciclo, a função remove elementos de uma min heap (quase) cheia, e insere numa árvore (quase) vazia. Nas iterações finais os papéis invertem-se, remove-se de uma heap (quase) vazia e insere-se numa árvore (quase) cheia. Logo em média teremos um custo logarítmico em cada iteração. São efetuadas N iterações, logo o custo da função `heap2AVL()` será $\Theta(N \times \log_{(} N))$.

Exercício 35

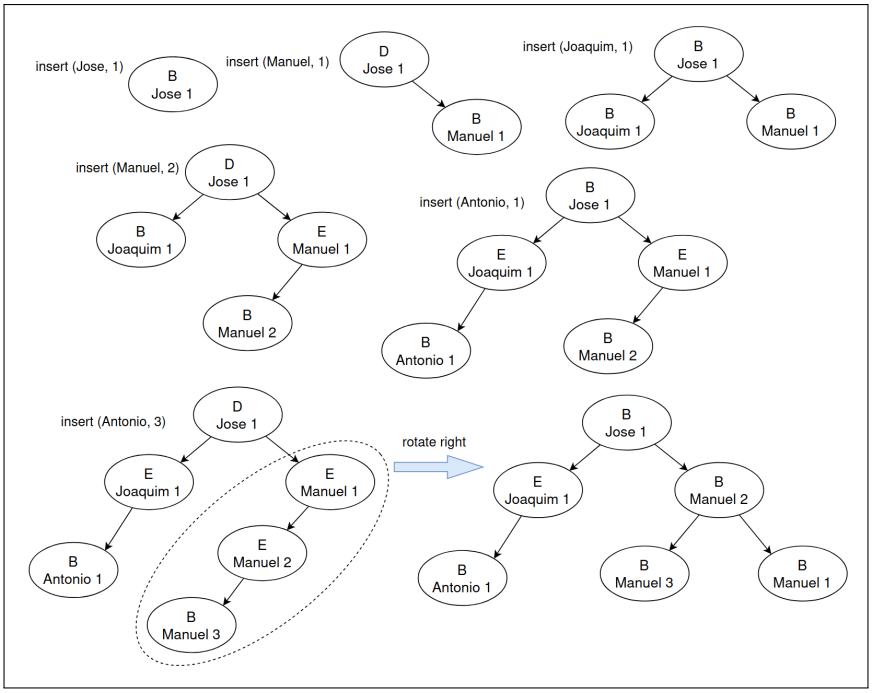

Figura 16: Exercício 35-1

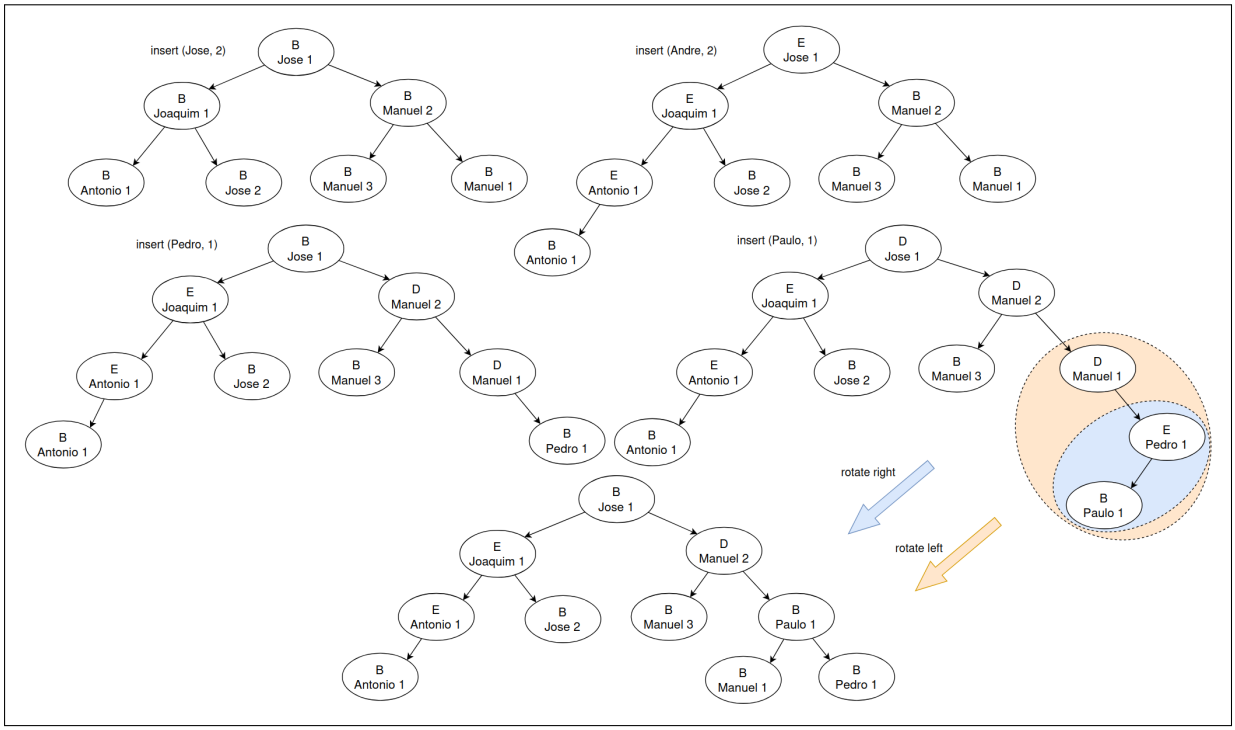

Figura 17: Exercício 35-2

```c
int allCopies(Dictionary dic, char key[11]) {
    if (dic == NULL) return 0;

    int count = 0;
    int comp = strcmp(dic->key, key);

    if (comp == 0) {
        count++;
        count += allCopies(dic->left, key);
        count += allCopies(dic->right, key);
    } else if (comp < 0)
        count += allCopies(dic->right, key);
    else
        count += allCopies(dic->left, key);

    return count;
}
```

Exercício 36

```c
int heapOKRec(BTree arv, int value) {
    if (arv == NULL) return 1;
    if (value > arv->value) return 0;
    return heapOKRec(arv->left, arv->value) &&
           heapOKRec(arv->right, arv->value);
}

int heapOK(BTree a) {
    return a == NULL || heapOKRec(a, a->value);
}
```

Exercício 37

```c
#define PARENT(i) (i - 1) / 2
#define LEFT(i) 2 * i + 1
#define RIGHT(i) 2 * i + 2
void swap(int * array, int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

void bubbleUp(int * array, int i) {
    int parent = PARENT(i);
    while (i > 0 && array[parent] > array[i]) {
        swap(array, i, parent);
        i = parent;
        parent = PARENT(parent);
    }
}
```

```c
void insertHeap(Heap *h, int x) {
    if (h->used == h->size) {
        h->size *= 2;
        h->heap = realloc(h->heap, h->size * sizeof(int));
    }

    h->heap[h->used] = x;
    bubbleUp(h->heap, h->used++);
}
```

Exercício 38

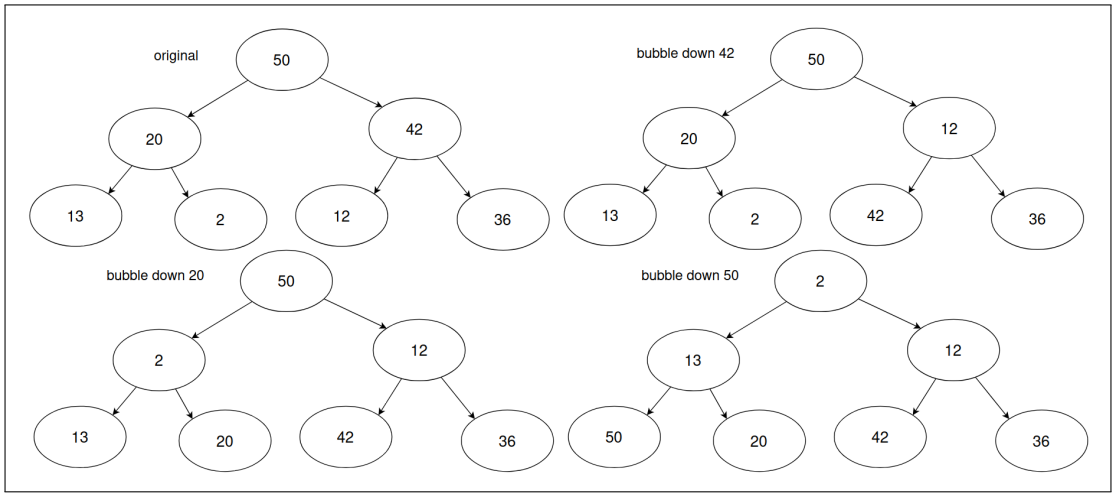

Figura 18: Invocação `heapify()`

$$T_{heapify}(N) =$$

Exercício 39

$$T_{deepest}(N) = \begin{cases} 0 & N \leq 0 \\ 1 + 2 \times T(N/2) & N > 0 \end{cases} = \sum_{i=0}^{1+\log_2(N)} 2^i = 4 \times N - 1 = \Theta(N)$$

```c
int deepest(AVLTree *a) {
    if (*a == NULL) return 0;
    int height = 1;
    while (*a != NULL && ((*a)->left != NULL || (*a)->right != NULL)) {
        a = (*a)->bal == Right ? &((*a)->right) : &((*a)->left);
        height++;
    }
    return height;
}
```

$$T_{deepest}(N) = \Theta(\log_2(N))$$

Exercício 40

```c
int fromChain(HashChain h1, HashOpen h2) {
    // estou a assumir que h2 já está inicializada com Livre
    int i, index;
    struct lista * temp = NULL;
    for (i = 0; i < HSIZE; i++) {
        temp = h1[i];
        index = i;
        while (temp != NULL) {
            while (h2[index].estado == 1)
                index = (index + 1) % HSIZE;

            h2[index].estado = 1;
            h2[index].valor = temp->valor;

            temp = temp->prox;
        }
    }
    return 0;
}
```

```c
#define CHAIN_SIZE ...
#define OPEN_SIZE ...
int fromChain(HashChain h1, HashOpen h2) {
    // estou a assumir que h2 já está inicializada com Livre
    int i, index;
    struct lista * temp = NULL;
    for (i = 0; i < CHAIN_SIZE; i++) {
        temp = h1[i];
        while (temp != NULL) {
            index = hash(temp->valor, OPEN_SIZE);
            while (h2[index].estado == 1)
                index = (index + 1) % OPEN_SIZE;

            h2[index].estado = 1;
            h2[index].valor = temp->valor;

            temp = temp->prox;
        }
    }
    return 0;
}
```

Exercício 41

	0	1	2	3	4	5	6	7	8	9	10	11	12
add 40, 80, 60, 260	Used	Used	Used	Free	Free	Free	Free	Free	Used	Free	Free	Free	Free
	260	40	80						60				
	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12
add 54	Used	Used	Used	Used	Free	Free	Free	Free	Used	Free	Free	Free	Free
	260	40	80	54					60				
	0	0	0	1	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12
add 65	Used	Used	Used	Used	Used	Free	Free	Free	Used	Free	Free	Free	Free
	260	40	80	54	65				60				
	0	0	0	1	4	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12
add 140	Used	Used	Used	Used	Used	Free	Free	Free	Used	Free	Used	Free	Free
	260	40	80	54	65				60		140		
	0	0	0	1	4	0	0	0	0	0	0	0	0

Figura 19: Inserir 40, 80, 60, 260, 54, 65, 140

Exercício 42

```
AVL maisProfundo(AVL a) {
    while (a != NULL && (a->left != NULL || a->right != NULL))
        a = a->bal == RIGHT ? a->right : a->left;

    return a;
}
```

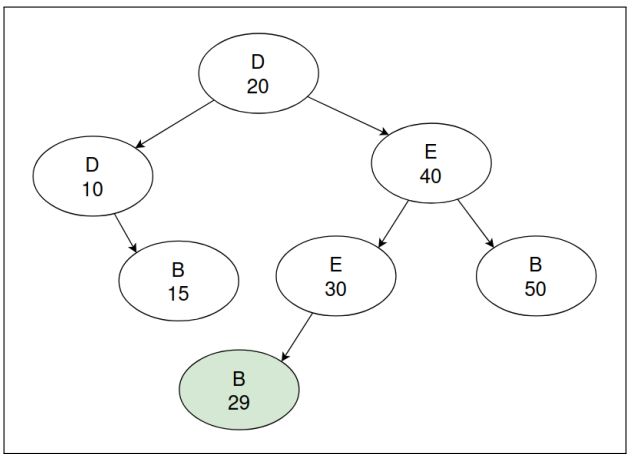

Figura 20: Árvore AVL

O resultado da função `maisProfundo()` aplicada à árvore apresentada será o nodo 29, presente no 4º nível da árvore.

Exercício 43

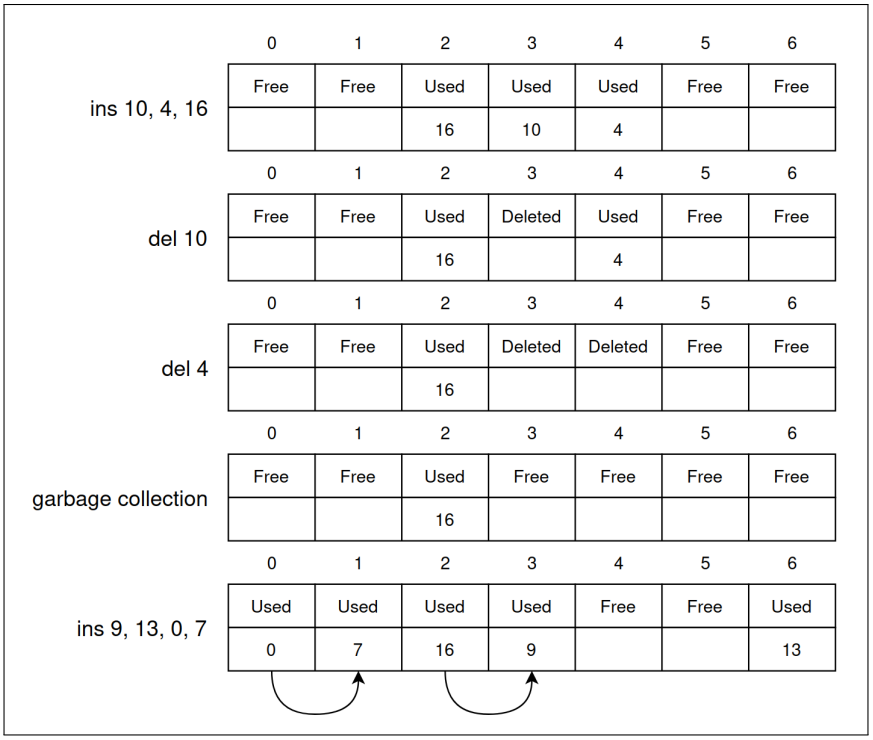

Figura 21: Hash Table com Linear Probing

Exercício 44

```
AVL fromListRec(LInt *l, int n, int *height) {
    if (n == 0) {
        *height = 0;
        return NULL;
    }

    int hL, hR, middle = n / 2;
    AVL left = fromListRec(l, middle, &hL);
    AVL root = malloc(sizeof(struct avl));
    root->value = (*l)->value;
    root->left = left;
    *l = (*l)->next;

    AVL right = fromListRec(l, n - middle - 1, &hR);
    root->right = right;
    root->bal = hR - hL;
    *height = (hL > hR ? hL : hR) + 1;

    return root;
}
AVL fromList (LInt l, int n) {
    int h;
    return fromListRec(&l, n, &h);
}
```

Exercício 45

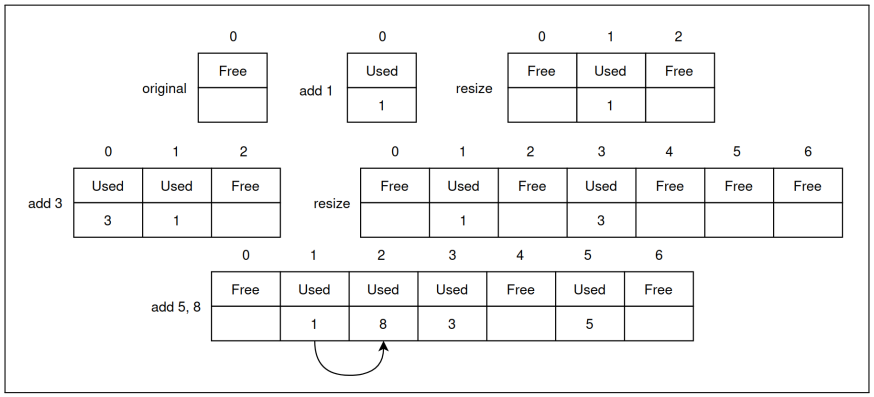

Figura 22: Inserir 3, 5, 8

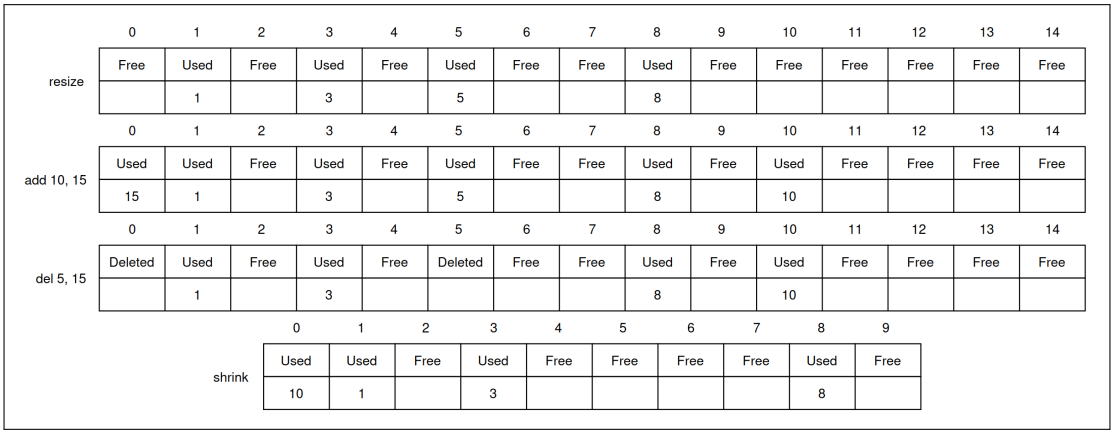

Figura 23: Inserir 10, 15 e Remover 5, 15

Exercício 46

```
void shrinkTHash(THash h) {
    int small_size = (h->tamanho * 2) / 3, index, i;
    struct entry * temp = calloc(small_size, sizeof(struct entry));
    for (i = 0; i < small_size; i++)
        temp[i].status = LIVRE;
    for (i = 0; i < h->tamanho; i++) {
        if (h->tabela[i].status == OCUPADO) {
            index = hash(h->tabela[i].value, small_size);
            while (temp[index].status == OCUPADO)
                index = (index + 1) % small_size;
            temp[index].status = OCUPADO;
            temp[index].value = h->tabela[i].value;
        }
    }
    free(h->tabela);
    h->tabela = temp;
    h->tamanho = small_size;
}
```

```c
void removeTHash(THash h, int key) {
    int index = hash(key, h->tamanho);
    int count = 0;

    while (h->tabela[index].status != LIVRE && count < h->tamanho) {
        if (h->tabela[index].status == OCUPADO &&
            h->tabela[index].value == key) {
            h->tabela[index].status = APAGADO;
            h->ocupados--;
            break;
        }
        index = (index + 1) % h->tamanho;
        count++;
    }

    if ((h->ocupados / (float) h->tamanho) < 0.33)
        shrinkTHash(h);
}
```

Exercício 47

```c
AVL rotateRight (AVL a) {
    AVL left = a->esq;

    a->esq = left->dir;
    left->dir = a;

    return left;
}
```

$$T_{spine}(N) = \begin{cases} 0 & N \leq 0 \\ 1 + T(N-1) & N > 0 \end{cases} = \sum_{i=1}^{N} 1 = \Theta(N)$$

Como a árvore já está em espinha, todos os nodos terão `a->esq == NULL`, logo o ciclo nunca será executado. Nesta análise estou a contar o número de vezes que a condição do ciclo é avaliada, que corresponde a uma vez por chamada recursiva.