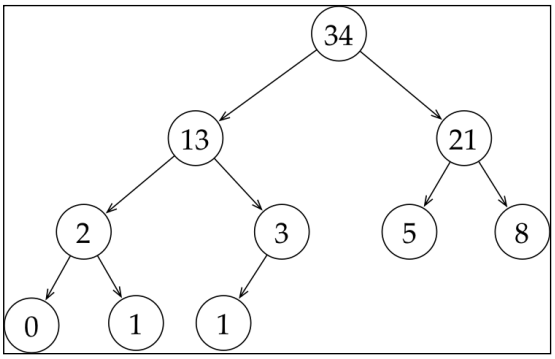
Estrutura de Dados 2 - 08.06.22

Nome: Kezia Campos

TAREFA E INDICAÇÃO DE LEITURA - ÁRVORES ESPECIAIS (ESTUDO DIRIGIDO)

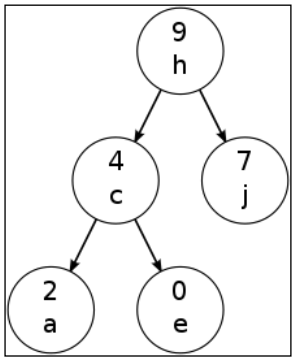
Parte 1 - Exercícios Heaps e Treaps (tipos especiais de Árvores Binárias)

1) Explique e diferencie Heaps e Treaps.



Heap são estruturas de dados baseado em árvores binárias de dados quase completa, não sendo árvores de binárias de pesquisa pois o valor de seu nó é maior ou igual ao valor de seus filhos.

Fonte: Algoritmo em Python

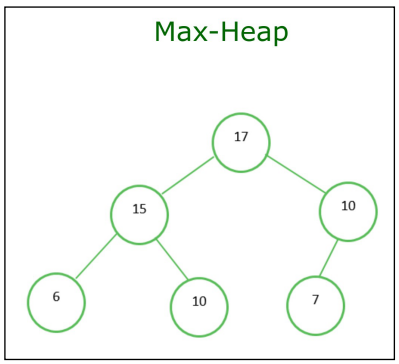


Treap é a **árvore de busca binária aleatória** onde possui um conjunto de chaves dinâmicas e ordenadas.

É basicamente a junção de **árvore** com **heap** (treap = tree + heap).

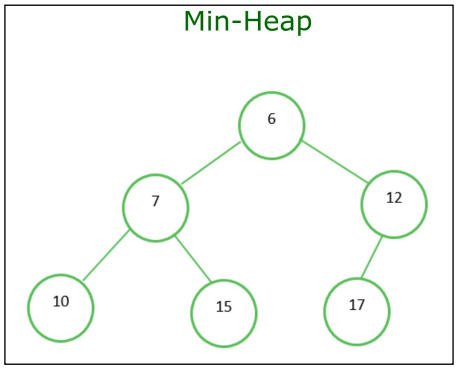
Fonte: Wikipedia

2) O que são Heaps máxima e mínima? Explique cada qual por meio de um exemplo.



Heap Máximo:
É quando o valor de todos os nós são menores que os de seus respectivos pais, como é possível observar no exemplo ao lado.

Fonte: Acervo Lima



Heap Mínimo:

É quando o valor de todos os nós são maiores que os de seus respectivos pais.

Fonte: Acervo Lima

3) Considere na Figura 1 a ordem de inserção de dados no array correspondente a Heap máxima. Logo, faça:

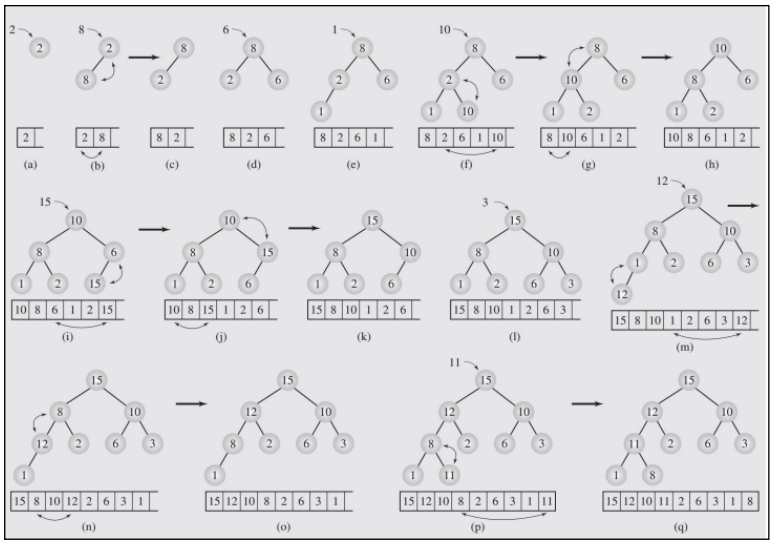


Figura 1

a) Explique como se dá a inserção cuja organização obedece ao método cima para-baixo?

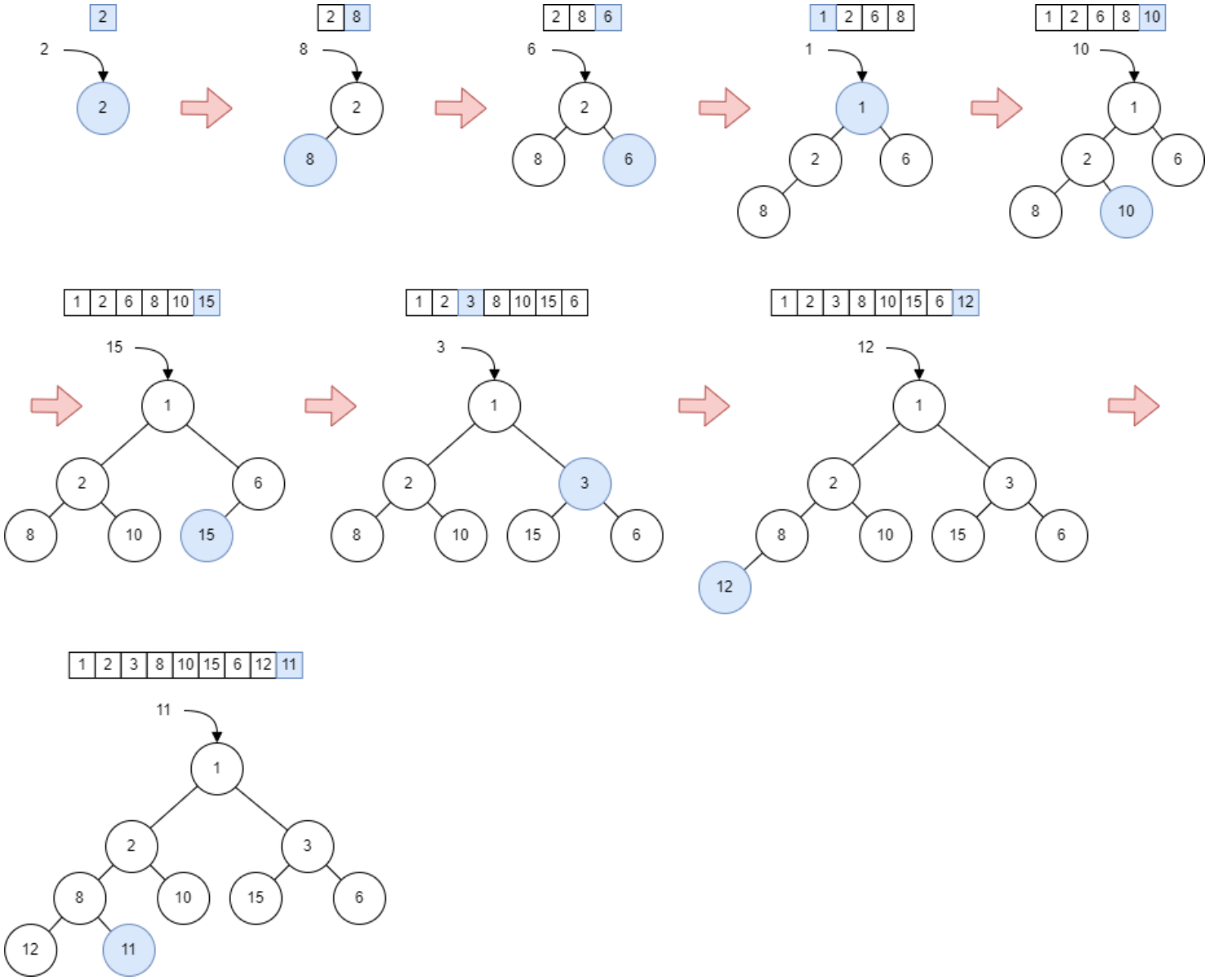
A inserção segue a seguinte regra "***Cada nó possui prioridade maior do que seus dois filhos, sendo o elemento de maior prioridade é sempre a raiz da árvore***".

```c
int pai(int i)
{
    return (i/2);
}
int esq(int i)
{
    return (i*2);
}
int dir(int i)
{
    return (i*2+1);
}
```

Para um determinado elemento **i** (**i** a posição do elemento de inserção):

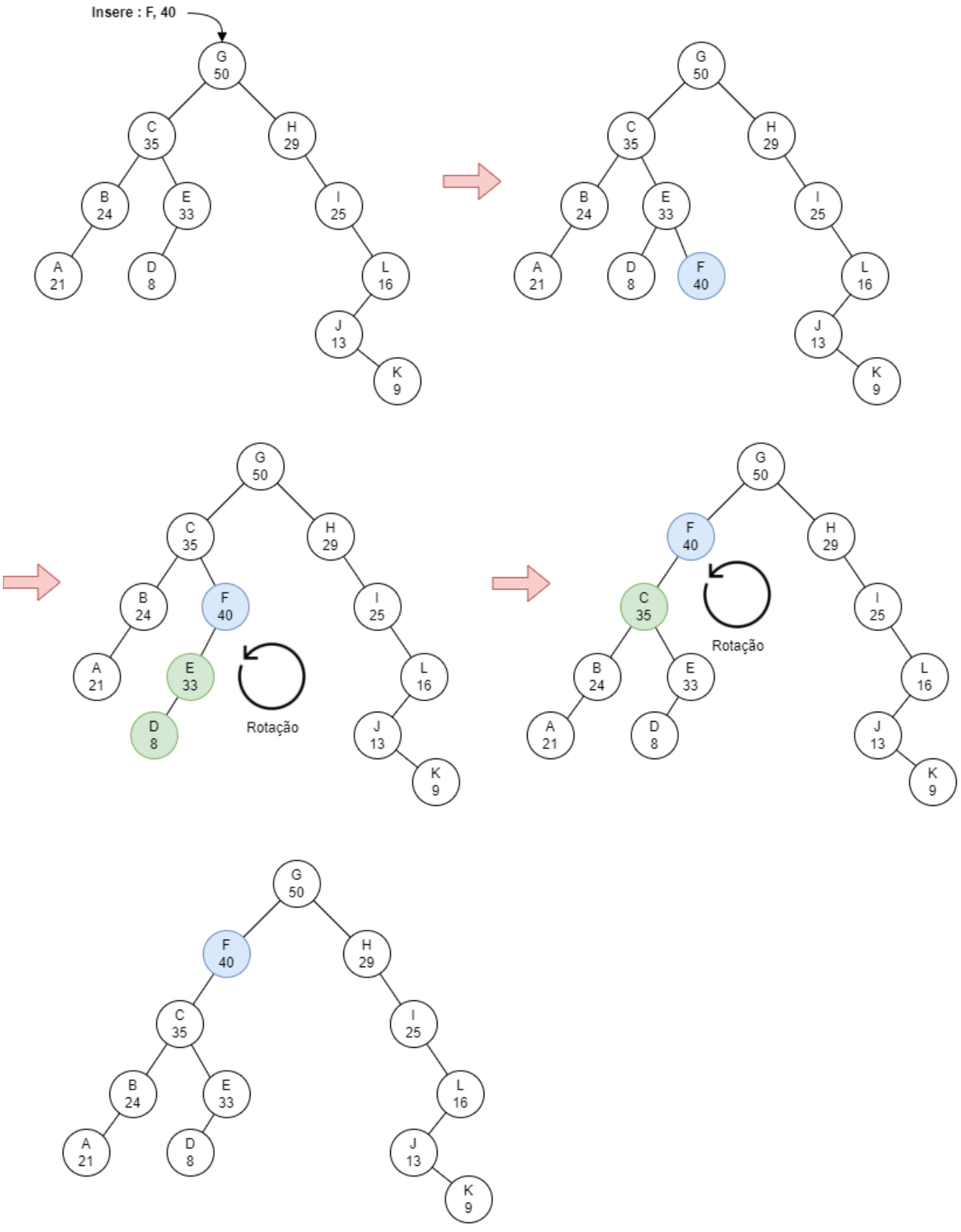
- o pai de i é i/2
- filho esquerdo é i * 2
- filho direito é i * 2 + 1

b) Supondo a mesma sequência de inserção, demonstre como seria formar uma Heap mínima pelo mesmo método cima-para-baixo?

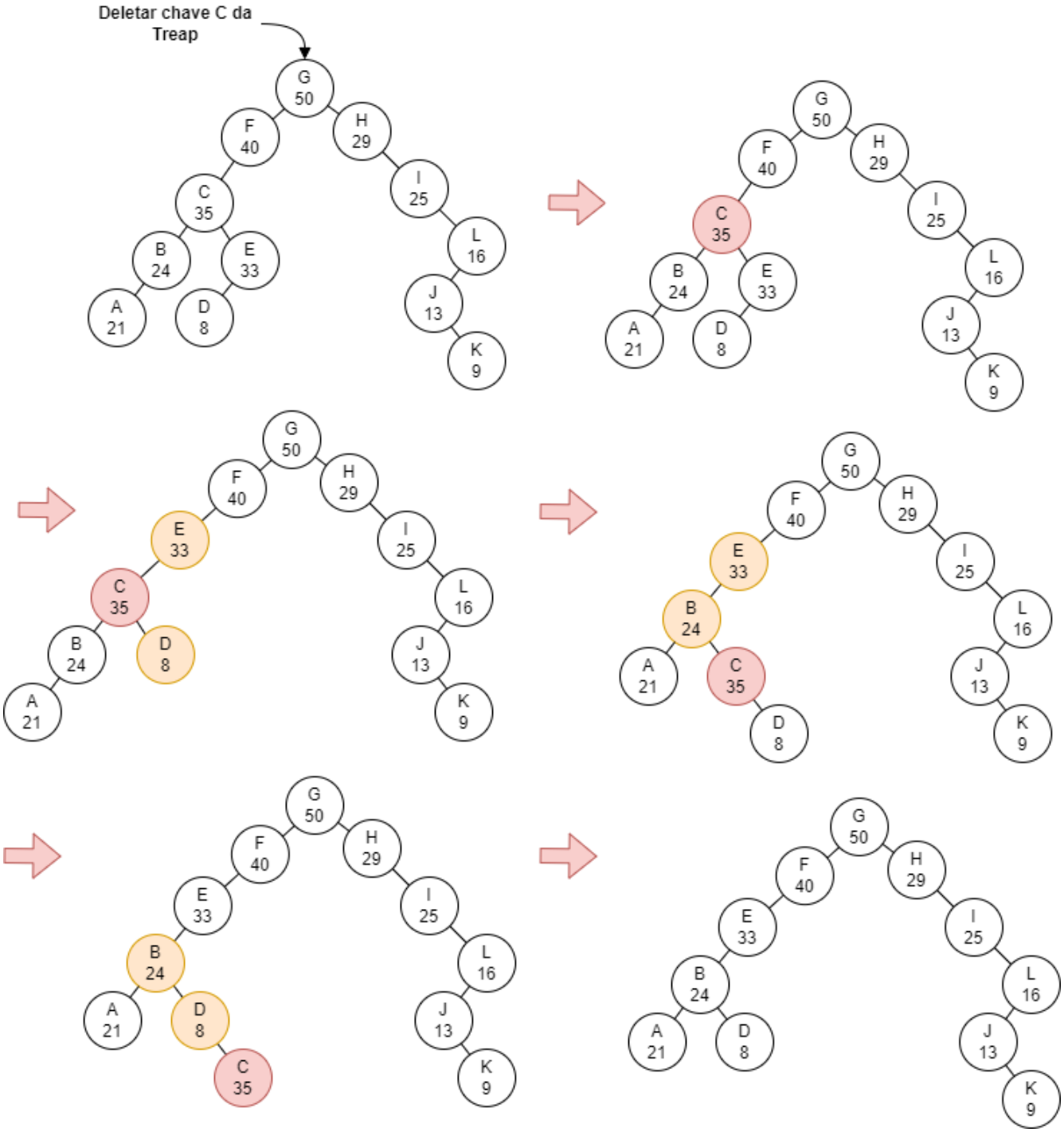


4) Demonstre uma Treap e como se dá a inserção e remoção de dados em um exemplo (dica: leitura recomendada do livro Adam Drozdek, Seção 6.10).

TREAP - Inserção



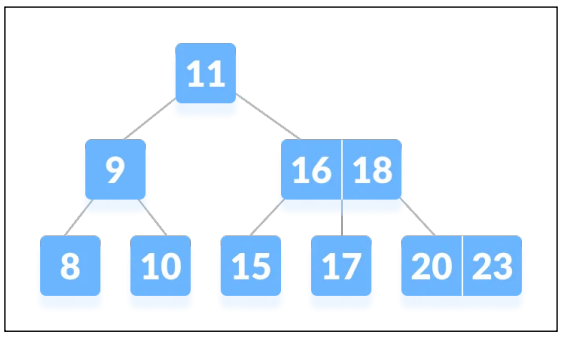
TREAP - Remoção



Parte 2 – Exercícios sobre Árvores múltiplas B, B* e B+

Desenvolva um relatório promovendo um paralelo entre os tipos de árvores múltiplas citadas. Neste, responda: Que aplicações/diferenças ocorrem na estruturação?
Paute suas explicações em exemplos (exemplo com codificação em linguagem C é opcional, mas será muito bem visto nas correções).

Árvore Múltipla B



Fonte: Tree B

A **árvore B** é um tipo especial de árvore de busca auto-balanceada na qual cada nó pode conter ***mais de uma chave*** e ***pode ter mais de dois filhos***. É uma forma generalizada da árvore de busca binária . É otimizada para acesso a grandes volumes de dados em disco e por isso é muito utilizada para recuperação de dados.

Exemplo em C de Árvore Múltipla B

Criação do Nó da Árvore

```c
//Inclui todas as bibliotecas padrão
#include <bits/stdc++.h>

#define max 3
#define min 2

struct No
{
  int val[max + 1], contador;
  struct No *link[max + 1];
};

struct No *raiz;

// Criando No
struct No *criandoNo(int val, struct No *filho)
{
  struct No *newNo;
  newNo = (struct No *)malloc(sizeof(struct No));
  newNo->val[1] = val;
  newNo->contador = 1;
  newNo->link[0] = raiz;
  newNo->link[1] = filho;
  return newNo;
}
```

```c
// Inserindo NO
void insereNo(int val, int pos, struct No *no, struct No *filho)
{
  int j = no->contador;
  while (j > pos) {
    no->val[j + 1] = no->val[j];
    no->link[j + 1] = no->link[j];
    j--;
  }
  no->val[j + 1] = val;
  no->link[j + 1] = filho;
  no->contador++;
}
```

Operação de Inserção

1) Se a árvore estiver vazia, aloque um nó raiz e insira a chave.

2) Atualize o número permitido de chaves no nó.

3) Pesquise o nó apropriado para inserção.

4) Se o nó estiver cheio, siga as etapas abaixo (5)

```c
// Divide No
void divideNo(int val, int *pval, int pos, struct No *no, struct No *filho, struct No **newNo)
{
  int med, j;

  if (pos > min)
    med = min + 1;
  else
    med = min;

  *newNo = (struct No *)malloc(sizeof(struct No));
  j = med + 1;
  while (j <= max) {
    (*newNo)->val[j - med] = no->val[j];
    (*newNo)->link[j - med] = no->link[j];
    j++;
  }
  no->contador = med;
  (*newNo)->contador = max - med;

  if (pos <= min) {
    insereNo(val, pos, no, filho);
  } else {
    insereNo(val, pos - med, *newNo, filho);
  }
  *pval = no->val[no->contador];
  (*newNo)->link[0] = no->link[no->contador];
  no->contador--;
}
```

5) Insira os elementos em ordem crescente.

6) Se existem elementos maiores que seu limite. Então, divida na mediana.

7) Empurre a chave da mediana para cima e faça as chaves esquerdas como filho esquerdo e as chaves direitas como filho direito.

8) Se o nó não estiver cheio, siga as etapas abaixo (9).

9) Insira o nó em ordem crescente.

```c
// Definindo o valor
int setvalor(int val, int *pval,
             struct No *no, struct No **filho) {
  int pos;
  if (!no) {
    *pval = val;
    *filho = NULL;
    return 1;
  }

  if (val < no->val[1]) {
    pos = 0;
  } else {
    for (pos = no->contador;
       (val < no->val[pos] && pos > 1); pos--)
      ;
    if (val == no->val[pos]) {
      printf("Duplicatas não são permitidas\n");
      return 0;
    }
  }
  if (setvalor(val, pval, no->link[pos], filho)) {
    if (no->contador < max) {
      insereNo(*pval, pos, no, *filho);
    } else {
      divideNo(*pval, pval, pos, no, *filho, filho);
      return 1;
    }
  }
  return 0;
}
```

```c
// Inserindo o valor
void insert(int val) {
  int flag, i;
  struct No *filho;

  flag = setvalor(val, &i, raiz, &filho);
  if (flag)
    raiz = criandoNo(i, filho);
}
```

```c
// Busca no
void busca(int val, int *pos, struct No *myNo) {
  if (!myNo) {
    return;
  }

  if (val < myNo->val[1]) {
    *pos = 0;
  } else {
    for (*pos = myNo->contador;
       (val < myNo->val[*pos] && *pos > 1); (*pos)--)
      ;
    if (val == myNo->val[*pos]) {
      printf("%d foi encontrado!", val);
      return;
    }
  }
  busca(val, pos, myNo->link[*pos]);

  return;
}
```

```c
// Traversal do no
void traversal(struct No *myNo) {
  int i;
  if (myNo) {
    for (i = 0; i < myNo->contador; i++) {
      traversal(myNo->link[i]);
      printf("%d ", myNo->val[i + 1]);
    }
    traversal(myNo->link[i]);
  }
}
```

```c
int main() {
  int val, ch;

  insert(8);
  insert(9);
  insert(10);
  insert(11);
  insert(15);
  insert(16);
  insert(17);
  insert(18);
  insert(20);
  insert(23);

  traversal(raiz);

  printf("\n");
  busca(11, &ch, raiz);
}
```

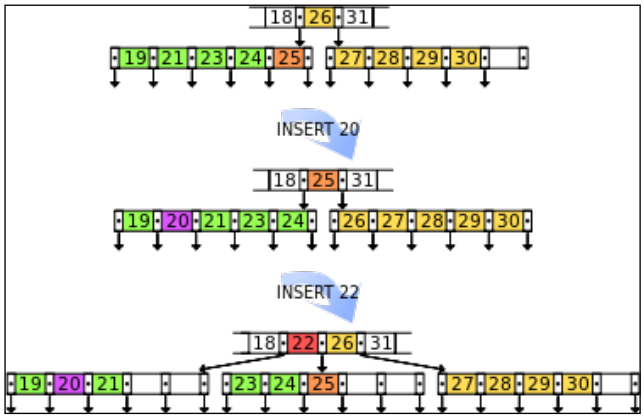
Inserindo no programa dados para execução e em seguida buscando pela chave 11.

Resultado:

```
Run:  ArvoreB
"C:\Users\kezia\Documents\GitHub\Femass_EstruturaDeDados_C\Estrutura de Dados 2\
8 9 10 11 15 16 17 18 20 23
11 foi encontrado!
Process finished with exit code 0
```

Árvore Múltipla B*

Uma árvore B* de ordem m apresenta as seguintes propriedades:



Fonte: Wikipedia

1.Cada página apresenta no máximo m páginas filhas

2.Uma página folha contém pelo menos **⌊(2m-1)/3⌋** chaves e no máximo **m-1**

3.Todas as páginas folha estão no mesmo nível

4.Toda página, exceto a raiz e as folhas possuem no máximo **(2m-1)/3** descendentes

5.Uma página não folha com **k** páginas filhas possui **k-1** chaves

Exemplo em C de Árvore Múltipla B*

Classe de criação do Nó da Árvore

```c
//Inclui todas as bibliotecas padrão
#include <bits/stdc++.h>
using namespace std;

#define N 4

struct no
{

    // chave de N-1 nos
    int chave[N - 1];
    struct no* filho[N];
    // se for folha, ehFolha=1 senão ehFolha=0
    int ehFolha;
    //Contador das chaves usadas no no
    int n;
    struct no* pai;
};
```

```
struct no* busca(struct no* raiz, int k, struct no* pai, int x)
{
    if (raiz)
    {
        // Se a raiz passada for uma folha no, então
        // k pode ser inserido neste próprio no
        if (raiz->ehFolha == 1)
            return raiz;

        // Se a raiz passada não for uma folha no,
        // implica que há um ou mais filhos
        else
        {
            int i;

            // Se passada a chave inicial da raiz é ela mesma
            // mais distante do que o elemento a ser inserido,
            // precisamos inserir uma nova folha à esquerda da raiz
            if (k < raiz->chave[0])
                raiz = busca(raiz->filho[0], k, raiz, 0);

            else
            {

                // Encontra a primeira chave cujo valor é maior
                // que o valor de inserção
                // e insere no filho dessa chave
                for (i = 0; i < raiz->n; i++)
                    if (raiz->chave[i] > k)
                        raiz = busca(raiz->filho[i], k, raiz, i);

                // Se todas as chaves forem menores que a inserção
                // valor da chave, inserido à direita da última chave
                if (raiz->chave[i - 1] < k)
                    raiz = busca(raiz->filho[i], k, raiz, i);
            }
        }
    }
    else {

        // Se a raiz passada for NULL (não existe tal
        // filho não para pesquisar), então crie uma nova folha
        // no nesse local
        struct no* novoNo = new struct no;
        novoNo->ehFolha = 1;
        novoNo->n = 0;
        pai->filho[x] = novoNo;
        novoNo->pai = pai;
        return novoNo;
    }
}
```

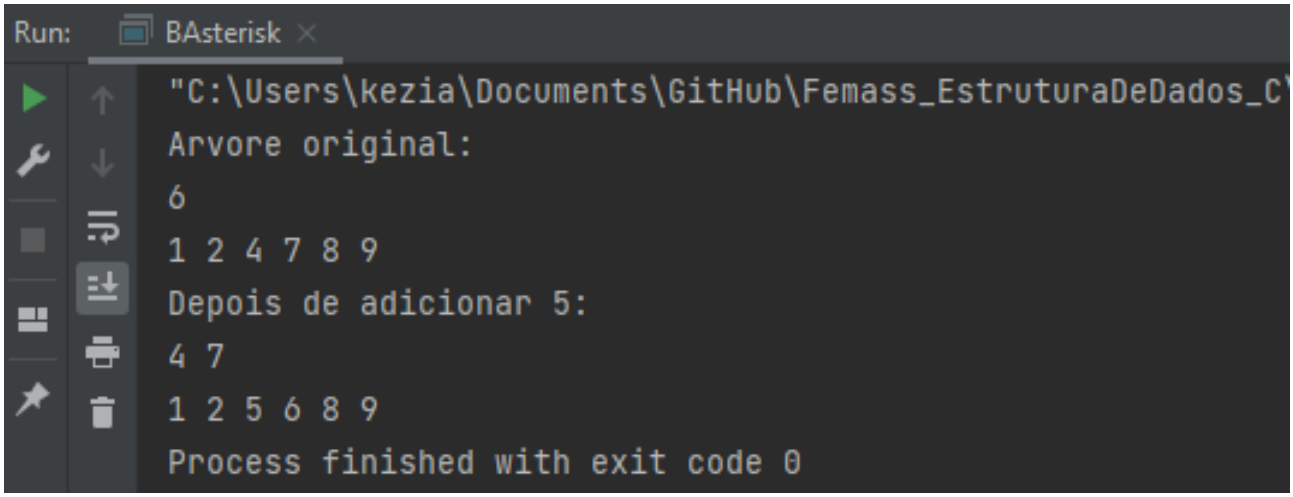
```c
struct no* insere(struct no* raiz, int k)
{
    if (raiz)
    {
        struct no* p = busca(raiz, k, NULL, 0);
        struct no* q = NULL;
        int e = k;

        // Se a folha não estiver vazia, simplesmente
        // adiciona o elemento e retorna
        for (int e = k; p; p = p->pai)
        {
            if (p->n == 0)
            {
                p->chave[0] = e;
                p->n = 1;
                return raiz;
            }

            // Se o número de chaves preenchidas for menor
            // que o máximo
            if (p->n < N - 1) {
                int i;
                for (i = 0; i < p->n; i++)
                {
                    if (p->chave[i] > e)
                    {
                        for (int j = p->n - 1; j >= i; j--)
                            p->chave[j + 1] = p->chave[j];
                        break;
                    }
                }
                p->chave[i] = e;
                p->n = p->n + 1;
                return raiz;
            }
```

```
// Se o número de chaves preenchidas for igual ao máximo
            // e não é raiz e tem espaço no pai
            if (p->n == N - 1 && p->pai && p->pai->n < N)
            {
                int m;
                for (int i = 0; i < p->pai->n; i++)
                    if (p->pai->filho[i] == p)
                    {
                        m = i;
                        break;
                    }
                // Se o irmão direito é possível
                if (m + 1 <= N - 1)
                {
                    //q é o irmão certo
                    q = p->pai->filho[m + 1];

                    if (q)
                    {
                        // Se o irmão direito estiver cheio
                        if (q->n == N - 1)
                        {
                            struct no* r = new struct no;
                            int* z = new int[((2 * N) / 3)];
                            int pai1, pai2;
                            int* marray = new int[2 * N];
                            int i;
                            for (i = 0; i < p->n; i++)
                                marray[i] = p->chave[i];
                            int fege = i;
                            marray[i] = e;
                            marray[i + 1] = p->pai->chave[m];
                            for (int j = i + 2; j < ((i + 2) + (q->n)); j++)
                                marray[j] = q->chave[j - (i + 2)];
                                // marray=bubblesort(marray, 2*N)
                                // uma implementação mais rigorosa
                                // ordena esses elementos

                            // Coloca primeiro (2*N-2)/3 elementos nas chaves de p
                            for (int i = 0; i < (2 * N - 2) / 3; i++)
                                p->chave[i] = marray[i];
                            pai1 = marray[(2 * N - 2) / 3];

                            // Coloca os próximos (2*N-1)/3 elementos nas chaves de q
                            for (int j = ((2 * N - 2) / 3) + 1; j < (4 * N) / 3; j++)
                                q->chave[j - ((2 * N - 2) / 3 + 1)] = marray[j];
                            pai2 = marray[(4 * N) / 3];

                            // Coloca os últimos (2*N)/3 elementos nas chaves de r
                            for (int f = ((4 * N) / 3 + 1); f < 2 * N; f++)
                                r->chave[f - ((4 * N) / 3 + 1)] = marray[f];
```

```
 // Como m=0 e m=1 são filhos da mesma chave,
                            // um caso especial é feito para eles
                            if (m == 0 || m == 1) {
                                p->pai->chave[0] = pai1;
                                p->pai->chave[1] = pai2;
                                p->pai->filho[0] = p;
                                p->pai->filho[1] = q;
                                p->pai->filho[2] = r;
                                return raiz;
                            }

                            else {
                                p->pai->chave[m - 1] = pai1;
                                p->pai->chave[m] = pai2;
                                p->pai->filho[m - 1] = p;
                                p->pai->filho[m] = q;
                                p->pai->filho[m + 1] = r;
                                return raiz;
                            }
                        }
                    }

                    // Se o irmão direito não estiver cheio
                    else
                    {
                        int put;
                        if (m == 0 || m == 1)
                            put = p->pai->chave[0];
                        else
                            put = p->pai->chave[m - 1];
                        for (int j = (q->n) - 1; j >= 1; j--)
                            q->chave[j + 1] = q->chave[j];
                        q->chave[0] = put;
                        p->pai->chave[m == 0 ? m : m - 1] = p->chave[p->n - 1];
                    }
                }
            }
        }
    }
    else
    {
        // Cria um novo no se raiz for NULL
        struct no* raiz = new struct no;
        raiz->chave[0] = k;
        raiz->ehFolha = 1;
        raiz->n = 1;
        raiz->pai = NULL;
    }
}
```

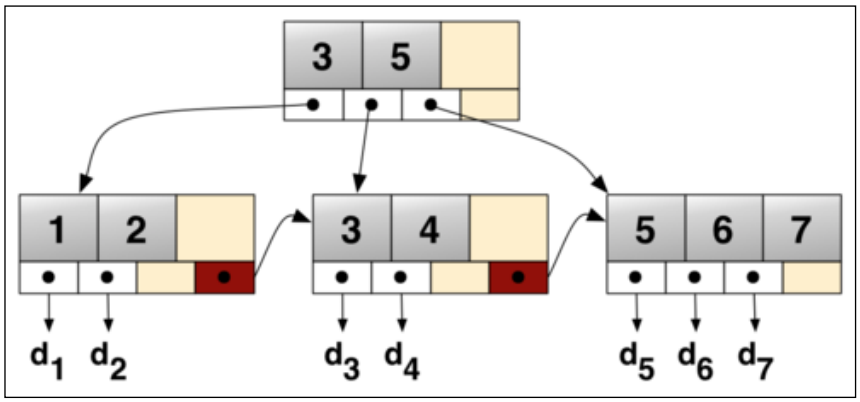
```
int main()
{
    // Começa com uma raiz vazia
    struct no* raiz = NULL;
    // insere 6
    raiz = insere(raiz, 6);

    // insere 1, 2, 4 à esquerda de 6
    raiz->filho[0] = insere(raiz->filho[0], 1);
    raiz->filho[0] = insere(raiz->filho[0], 2);
    raiz->filho[0] = insere(raiz->filho[0], 4);
    raiz->filho[0]->pai = raiz;

    // insere 7, 8, 9 à direita de 6
    raiz->filho[1] = insere(raiz->filho[1], 7);
    raiz->filho[1] = insere(raiz->filho[1], 8);
    raiz->filho[1] = insere(raiz->filho[1], 9);
    raiz->filho[1]->pai = raiz;

    cout << "Arvore original: " << endl;
    for (int i = 0; i < raiz->n; i++)
        cout << raiz->chave[i] << " ";
    cout << endl;
    for (int i = 0; i < 2; i++) {
        cout << raiz->filho[i]->chave[0] << " ";
        cout << raiz->filho[i]->chave[1] << " ";
        cout << raiz->filho[i]->chave[2] << " ";
    }
    cout << endl;


    cout << "Depois de adicionar 5: " << endl;

    raiz->filho[0] = insere(raiz->filho[0], 5);

    // Imprimindo os nos
    for (int i = 0; i <= raiz->n; i++)
        cout << raiz->chave[i] << " ";
    cout << endl;
    for (int i = 0; i < N - 1; i++)
    {
        cout << raiz->filho[i]->chave[0] << " ";
        cout << raiz->filho[i]->chave[1] << " ";
    }

    return 0;
}
```

Resultado:



Árvore Múltipla B+

Uma árvore B+ é uma forma avançada de uma árvore auto-balanceada na qual todos os valores estão presentes no nível folha.



Fonte: Wikipedia

Propriedades de uma **árvore B+**:

1. Todas as folhas estão no mesmo nível.
2. A raiz tem pelo menos dois filhos.
3. Cada nó, exceto a raiz, pode ter no máximo **m** filhos e pelo menos **m/2** filhos.
4. Cada nó pode conter no máximo **m - 1** chaves e um mínimo de **⌈m/2⌉ - 1** chaves.

Exemplo em C++ de Árvore Múltipla B+

```
//Inclui todas as bibliotecas padrão
#include <bits/stdc++.h>

using namespace std;
int MAX = 3;

// BP No
class No {
  bool ehFolha;
  int *chave, tam;
  No **ptr;
  friend class Arvore;

   public:
  No();
};

// BP tree
class Arvore {
  No *raiz;
  void insereInt(int, No *, No *);
  No *buscaPai(No *, No *);

   public:
  Arvore();
  void busca(int);
  void insere(int);
  void imprime(No *);
  No *getRaiz();
};

No::No() {
  chave = new int[MAX];
  ptr = new No *[MAX + 1];
}

Arvore::Arvore() {
  raiz = NULL;
}
```

```
// Busca
void Arvore::busca(int x) {
  if (raiz == NULL) {
    cout << "Arvore esta vazia\n";
  } else {
    No *cursor = raiz;
    while (cursor->ehFolha == false) {
      for (int i = 0; i < cursor->tam; i++) {
        if (x < cursor->chave[i]) {
          cursor = cursor->ptr[i];
          break;
        }
        if (i == cursor->tam - 1) {
          cursor = cursor->ptr[i + 1];
          break;
        }
      }
    }
    for (int i = 0; i < cursor->tam; i++) {
      if (cursor->chave[i] == x) {
        cout << "Encontrado!\n";
        return;
      }
    }
    cout << "Nao Encontrado!\n";
  }
}
```

```
// Insere
void Arvore::insere(int x) {
  if (raiz == NULL) {
    raiz = new No;
    raiz->chave[0] = x;
    raiz->ehFolha = true;
    raiz->tam = 1;
  } else {
    No *cursor = raiz;
    No *pai;
    while (cursor->ehFolha == false) {
      pai = cursor;
      for (int i = 0; i < cursor->tam; i++) {
        if (x < cursor->chave[i]) {
          cursor = cursor->ptr[i];
          break;
        }
        if (i == cursor->tam - 1) {
          cursor = cursor->ptr[i + 1];
          break;
```

```
        }
      }
    }
    if (cursor->tam < MAX) {
      int i = 0;
      while (x > cursor->chave[i] && i < cursor->tam)
        i++;
      for (int j = cursor->tam; j > i; j--) {
        cursor->chave[j] = cursor->chave[j - 1];
      }
      cursor->chave[i] = x;
      cursor->tam++;
      cursor->ptr[cursor->tam] = cursor->ptr[cursor->tam - 1];
      cursor->ptr[cursor->tam - 1] = NULL;
    } else {
      No *novaFolha = new No;
      int noVirtual[MAX + 1];
      for (int i = 0; i < MAX; i++) {
        noVirtual[i] = cursor->chave[i];
      }
      int i = 0, j;
      while (x > noVirtual[i] && i < MAX)
        i++;
      for (int j = MAX + 1; j > i; j--) {
        noVirtual[j] = noVirtual[j - 1];
      }
      noVirtual[i] = x;
      novaFolha->ehFolha = true;
      cursor->tam = (MAX + 1) / 2;
      novaFolha->tam = MAX + 1 - (MAX + 1) / 2;
      cursor->ptr[cursor->tam] = novaFolha;
      novaFolha->ptr[novaFolha->tam] = cursor->ptr[MAX];
      cursor->ptr[MAX] = NULL;
      for (i = 0; i < cursor->tam; i++) {
        cursor->chave[i] = noVirtual[i];
      }
      for (i = 0, j = cursor->tam; i < novaFolha->tam; i++, j++) {
        novaFolha->chave[i] = noVirtual[j];
      }
```

```
      if (cursor == raiz) {
        No *novaRaiz = new No;
        novaRaiz->chave[0] = novaFolha->chave[0];
        novaRaiz->ptr[0] = cursor;
        novaRaiz->ptr[1] = novaFolha;
        novaRaiz->ehFolha = false;
        novaRaiz->tam = 1;
        raiz = novaRaiz;
      } else {
        insereInt(novaFolha->chave[0], pai, novaFolha);
      }
    }
  }
}
```

```
// Insere Interno
void Arvore::insereInt(int x, No *cursor, No *filho) {
  if (cursor->tam < MAX) {
    int i = 0;
    while (x > cursor->chave[i] && i < cursor->tam)
      i++;
    for (int j = cursor->tam; j > i; j--) {
      cursor->chave[j] = cursor->chave[j - 1];
    }
    for (int j = cursor->tam + 1; j > i + 1; j--) {
      cursor->ptr[j] = cursor->ptr[j - 1];
    }
    cursor->chave[i] = x;
    cursor->tam++;
    cursor->ptr[i + 1] = filho;
  } else {
    No *novoInt = new No;
    int chaveVirtual[MAX + 1];
    No *virtualPtr[MAX + 2];
    for (int i = 0; i < MAX; i++) {
      chaveVirtual[i] = cursor->chave[i];
    }
    for (int i = 0; i < MAX + 1; i++) {
      virtualPtr[i] = cursor->ptr[i];
    }
    int i = 0, j;
    while (x > chaveVirtual[i] && i < MAX)
      i++;
    for (int j = MAX + 1; j > i; j--) {
      chaveVirtual[j] = chaveVirtual[j - 1];
    }
    chaveVirtual[i] = x;
    for (int j = MAX + 2; j > i + 1; j--) {
      virtualPtr[j] = virtualPtr[j - 1];
    }
```

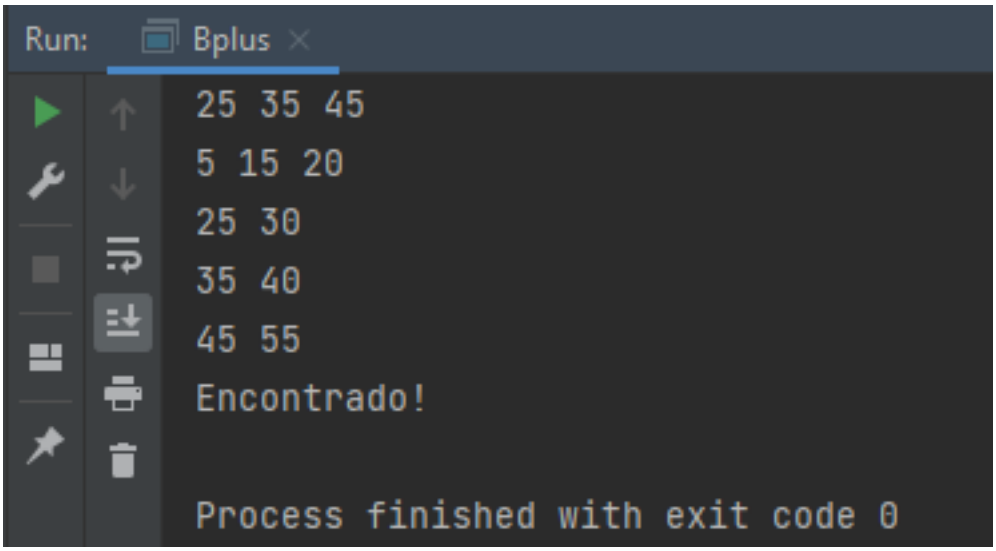
```
    virtualPtr[i + 1] = filho;
    novoInt->ehFolha = false;
    cursor->tam = (MAX + 1) / 2;
    novoInt->tam = MAX - (MAX + 1) / 2;
    for (i = 0, j = cursor->tam + 1; i < novoInt->tam; i++, j++) {
      novoInt->chave[i] = chaveVirtual[j];
    }
    for (i = 0, j = cursor->tam + 1; i < novoInt->tam + 1; i++, j++) {
      novoInt->ptr[i] = virtualPtr[j];
    }
    if (cursor == raiz) {
      No *novaRaiz = new No;
      novaRaiz->chave[0] = cursor->chave[cursor->tam];
      novaRaiz->ptr[0] = cursor;
      novaRaiz->ptr[1] = novoInt;
      novaRaiz->ehFolha = false;
      novaRaiz->tam = 1;
      raiz = novaRaiz;
    } else {
      insereInt(cursor->chave[cursor->tam], buscaPai(raiz, cursor), novoInt);
    }
  }
}
```

```
// Busca pai
No *Arvore::buscaPai(No *cursor, No *filho) {
  No *pai;
  if (cursor->ehFolha || (cursor->ptr[0])->ehFolha) {
    return NULL;
  }
  for (int i = 0; i < cursor->tam + 1; i++) {
    if (cursor->ptr[i] == filho) {
      pai = cursor;
      return pai;
    } else {
      pai = buscaPai(cursor->ptr[i], filho);
      if (pai != NULL)
        return pai;
    }
  }
  return pai;
}
```

```
// Imprime
void Arvore::imprime(No *cursor) {
  if (cursor != NULL) {
    for (int i = 0; i < cursor->tam; i++) {
      cout << cursor->chave[i] << " ";
    }
    cout << "\n";
    if (cursor->ehFolha != true) {
      for (int i = 0; i < cursor->tam + 1; i++) {
        imprime(cursor->ptr[i]);
      }
    }
  }
}

// Get raiz
No *Arvore::getRaiz() {
  return raiz;
}

int main() {
  Arvore No;
  No.insere(5);
  No.insere(15);
  No.insere(25);
  No.insere(35);
  No.insere(45);
  No.insere(55);
  No.insere(40);
  No.insere(30);
  No.insere(20);
  No.imprime(No.getRaiz());

  No.busca(15);
}
```

Inserindo no programa dados para execução e em seguida buscando pela chave 15.

Resultado:



Diferença entre estruturas de B, B* e B+:

*	Árvore B	Árvore B +	Árvore B *
1	Todos os nós internos e folha têm ponteiros de dados	Apenas nós folha têm ponteiros de dados	Cada nó contém, no mínimo, ⅔ do número máximo de chaves;
2	Uma vez que todas as chaves não estão disponíveis na folha, a pesquisa geralmente leva mais tempo.	Todas as chaves estão em nós folha, portanto, a pesquisa é mais rápida e precisa.	A subdivisão é adiada até que duas páginas irmãs estejam cheias.
3	Nenhuma duplicata de chaves é mantida na árvore.	A duplicata de chaves é mantida e todos os nós estão presentes na folha.	Todas as folhas aparecem no mesmo nível;
4	A inserção leva mais tempo e às vezes não é previsível.	A inserção é mais fácil e os resultados são sempre os mesmos.	Cada página possui um máximo de m descendentes;
5	A exclusão de um nó interno é muito complexa e a árvore deve passar por muitas transformações.	A exclusão de qualquer nó é fácil porque todos os nós são encontrados na folha.	Na sequência, a divisão do conteúdo das duas páginas em três páginas *(two-to-three split)* é realizada.
6	Os nós de folha não são armazenados como lista ligada estrutural.	Os nós de folha são armazenados como uma lista vinculada estrutural.	A raiz possui pelo menos 2 descendentes, a menos que seja um nó folha;
7	Nenhuma chave de pesquisa redundante está presente.	Chaves de pesquisa redundantes podem estar presentes.	A estratégia dessa variação é realizar o particionamento de duas páginas irmãs somente quando estas estiverem completamente cheias

Fonte: CamaraLeveger

Bibliografia

COMPUTER SCIENCE & ENGINEERING-UC SAN DIEGO. Lecture 6. Disponível em: <https://cseweb.ucsd.edu//~kube/cls/100/Lectures/lec6.treap/lec6.html>. Acesso em: 5 jun. 2022.

Diferença entre Min Heap e Max Heap – Acervo Lima. Disponível em: <https://acervolima.com/diferenca-entre-min-heap-e-max-heap/>. Acesso em: 2 jun. 2022.

DO ISEP, D. DE E. I. FILAS DE PRIORIDADE e HEAPS. Disponível em: <https://www.cin.ufpe.br/~afqa/Heaps.pdf>. Acesso em: 2 jun. 2022.

DROZDEK, A. Data structures and algorithms in C++ fourth edition. Disponível em: <http://itlectures.ro/wp-content/uploads/2016/04/AdamDrozdek__DataStructures_and_Algorithms_in_C_4Ed.pdf>. Acesso em: 4 jun. 2022.

GAZZOLA, M. G. Árvore B, B* e B+. Disponível em: <http://wiki.icmc.usp.br/images/8/8e/SCC578920131-B.pdf>. Acesso em: 5 jun. 2022.

GONÇALVES DE MOURA, A. J. Disponível em: <http://www.camaraleverger.mt.gov.br/painel/upload/COMPARA%C3%87%C3%83O%20DE%20ARVORES%20TIPO%20B.pdf>. Acesso em: 5 jun. 2022.