KAUNO TECHNOLOGIJOS UNIVERSITETAS

Duomenų struktūros

B-Tree

Technologinio projektinio darbo ataskaita

Atliko Benas Montvydas IFF 9-11

1.Užduotis:

Realizuokite vieną iš šių duomenų struktūrų. Pilnai ištestuokite realizuotos duomenų struktūros operacijas. Realizuotų duomenų struktūrų operacijų asimptotinis laiko ir atminties sudėtingumas turi atitikti konkrečios duomenų struktūros žinomą teorinį asimptotinį sudėtingumą.

5. B-Tree. Daugiau apie šią duomenų struktūrą: “Introduction to algorithms 3th edition” 18 skyrius. **(maks. 10 + 1 balas prie egzamino)**

2. B-Tree aprašymas:

B-Tree, tai savaime susibalansuojantis medis. Kuris skirtas apdoroti didelius kiekius duomenų. Palyginus su kitais medžiais, B-Tree turi labai greitą paiešką, nes visi duomenys surūšiuoti pagal tam tikrą parametrą ir ieškant informacijos nereikia pereiti per visus esančius duomenis. Tai pat, B-Tree aukštis yra palaikomas kuo minimalesnis, kad pagreitintų tam tikras operacijas.

Pagrindiniai metodai:

1. „Search“, algoritmo sudėtingumas O(logn)
2. „Insert“, algoritmo sudėtingumas O(logn)
3. „Delete“, algoritmo sudėtingumas O(logn)

Panaudosime formules apskaičiuoti:

t (Minimum degree) – turėtų būti pateiktas

1.Mažiausią aukštį:

$$h_{min} = \lceil \log_m(n+1) \rceil - 1$$

2.Didžiausią aukštį:

$$h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor$$

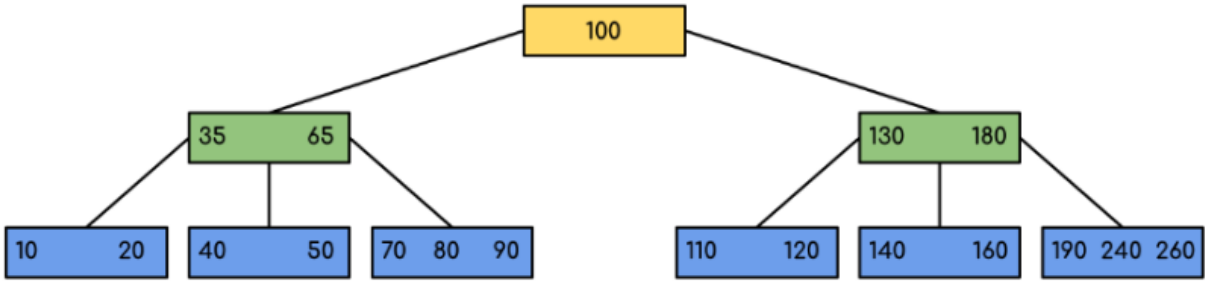
3.Daugiausia raktų, kiek gali būti:

maxKeys = 2t-1;

4.,,Vaikų“ kiekis:

Children = maxKeys +1;

B-Tree pavyzdys:



B-Tree pagrindinis aprašas:

1.BTree pagrindas:

```
public class BTree {
    //minimum degree;
    private int t;
    // The maximum number of keys in any node;
    private int maxKeys;
    // The root of the B-tree;
    private BNode root;

    public BTree(int t) {
        this.t = t;
        maxKeys = 2 * t - 1;
        root = new BNode( n: 0,  leaf: true); // root is a leaf
    }
```

2.BNode:

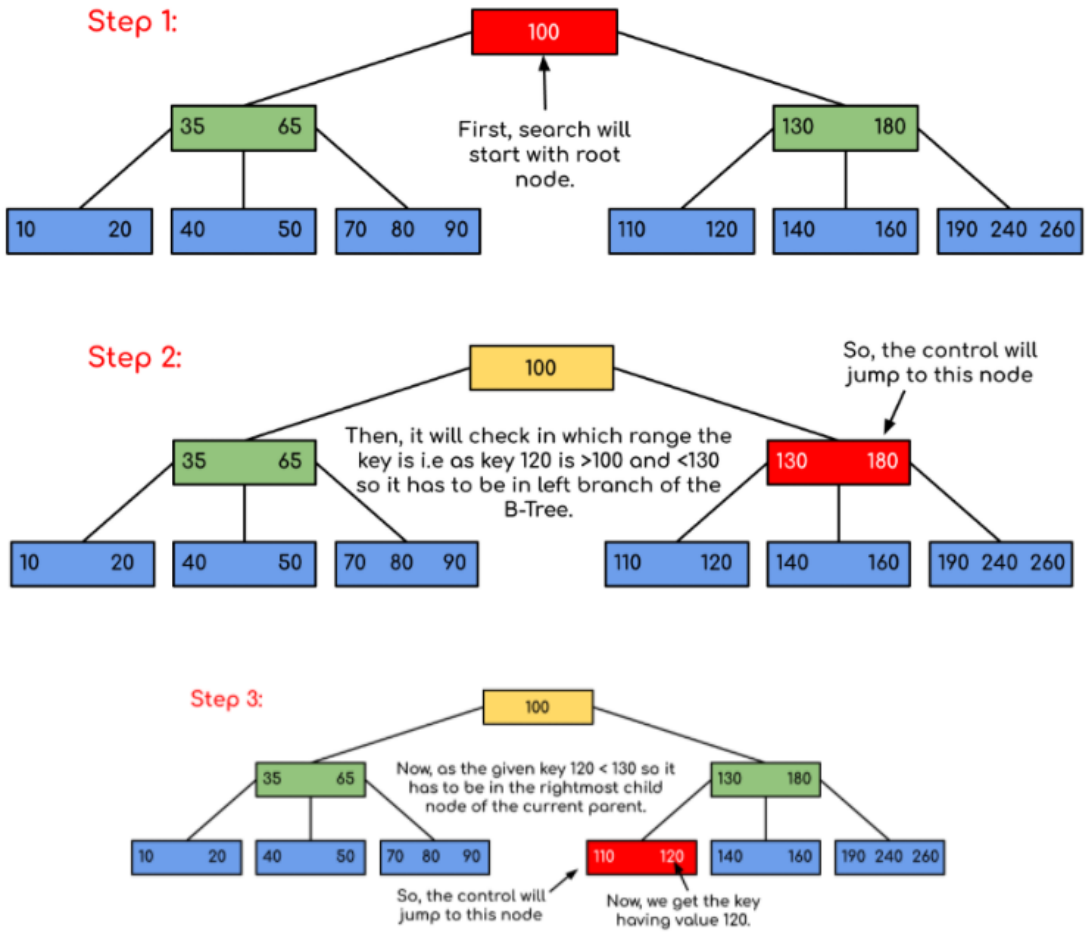
```
private class BNode {
    // Number of keys stored in a node
    public int n;
    public Integer[] keys;
    public BNode[] child;
    public boolean leaf;

    public BNode(int n, boolean leaf) {
        this.n = n;
        this.leaf = leaf;
        this.keys = new Integer[maxKeys];
        child = new BNode[maxKeys + 1];
    }
```

3.Pagrindinių metodų realizacija:

1.Search:

B-Tree paieškos metodas yra labai panašus, kaip ir „Binary Search Tree“ . Algoritmas labai panašus ir pasitelkia rekursija. Paiešką pradedam nuo kamieno ir einam rekursiškai į lapus, pasirinkdami tinkamą „vaiką“. Kai surandame tą reikšmę, mes grąžiname „Node“, o jei nerandame grąžiname „null“.

Vizualizacija:



Bendra realizacija:

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3      i = i + 1
4  if i ≤ x.n and k == x.key_i
5      return (x, i)
6  elseif x.leaf
7      return NIL
8  else DISK-READ(x.c_i)
9      return B-TREE-SEARCH(x.c_i, k)
```

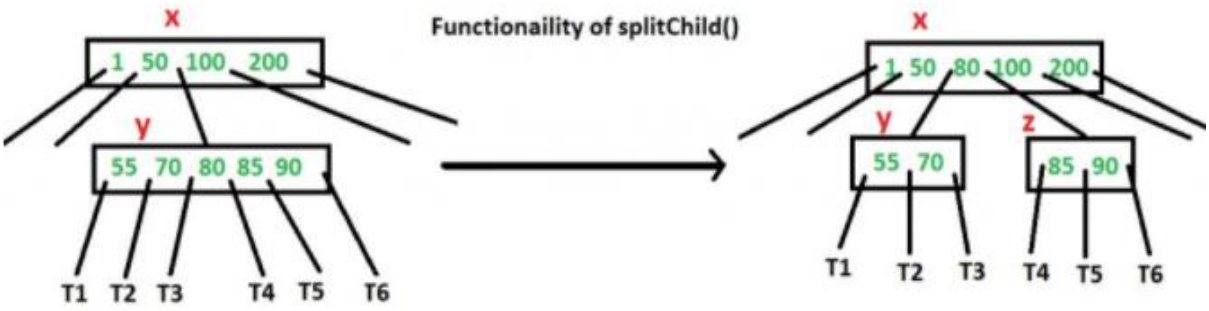
Mano sukurta realizacija:

```
public BNode Search(Integer k) {
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If the key is not found here and this is a leaf node
    if (leaf == true)
        return new BNode( n: 0,  leaf: true);

    // Go to the appropriate child
    return child[i].Search(k);
}
```

2.Insert:

B-Tree įterpimo metodą pradedame nuo kamieno ir einame link lapų. Kai pasiekiame tam tikrą lapą, mes įterpiame reikšmę, bet prieš įterpiant reikšmę, mes turime patikrinti, kad tam skirta vieta turi laisvos vietos. Jei laisvos vietos nėra, mes pasinaudojame operacija „splitChild()“. Ši operacija naudojama atskirti „vaiko“ reikšmes į dvi dalis, taip sukurdami papildamos vietos. Dėl šios operacijos medis gali augti ir į viršų.

Vizualizacija :



Bendra realizacija:

1.B-Tree splitChild:

```
B-TREE-SPLIT-CHILD(x, i)
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t − 1
 5  for j = 1 to t − 1
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t
 9          z.c_j = y.c_{j+t}
10  y.n = t − 1
11  for j = x.n + 1 downto i + 1
12      x.c_{j+1} = x.c_j
13  x.c_{i+1} = z
14  for j = x.n downto i
15      x.key_{j+1} = x.key_j
16  x.key_i = y.key_t
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

2.B-Tree insertNonFull:

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3      while i ≥ 1 and k < x.key_i
 4          x.key_{i+1} = x.key_i
 5          i = i − 1
 6      x.key_{i+1} = k
 7      x.n = x.n + 1
 8      DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10          i = i − 1
11      i = i + 1
12      DISK-READ(x.c_i)
13      if x.c_i.n == 2t − 1
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```

3.B-Tree insert:

```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t − 1
 3      s = ALLOCATE-NODE()
 4      T.root = s
 5      s.leaf = FALSE
 6      s.n = 0
 7      s.c_1 = r
 8      B-TREE-SPLIT-CHILD(s, 1)
 9      B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

Mano sukurta realizacija:

1.B-Tree SplitChild:

```
private void SplitChild(BNode x, int pos, BNode y) {
    BNode z = new BNode( n: 0,  leaf: true);
    z.leaf = y.leaf;
    z.n = t - 1;
    for (int j = 0; j < t - 1; j++) {
        z.keys[j] = y.keys[j + t];
    }
    if (!y.leaf) {
        for (int j = 0; j < t; j++) {
            z.child[j] = y.child[j + t];
        }
    }
    y.n = t - 1;
    for (int j = x.n; j >= pos + 1; j--) {
        x.child[j + 1] = x.child[j];
    }
    x.child[pos + 1] = z;

    for (int j = x.n - 1; j >= pos; j--) {
        x.keys[j + 1] = x.keys[j];
    }
    x.keys[pos] = y.keys[t - 1];
    x.n = x.n + 1;
}
```

2.B-Tree InsertNonFull:

```
private void InsertNonfull(BNode x, int k) {
    if (x.leaf) {
        int i = 0;
        for (i = x.n - 1; i >= 0 && k < x.keys[i]; i--) {
            x.keys[i + 1] = x.keys[i];
        }
        x.keys[i + 1] = k;
        x.n = x.n + 1;
    } else {
        int i = 0;
        for (i = x.n - 1; i >= 0 && k < x.keys[i]; i--) {
        }
        ;
        i++;
        BNode tmp = x.child[i];
        if (tmp.n == 2 * t - 1) {
            SplitChild(x, i, tmp);
            if (k > x.keys[i]) {
                i++;
            }
        }
        InsertNonfull(x.child[i], k);
    }
}
```

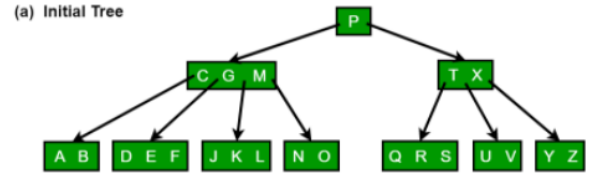
3.B-Tree Insert:

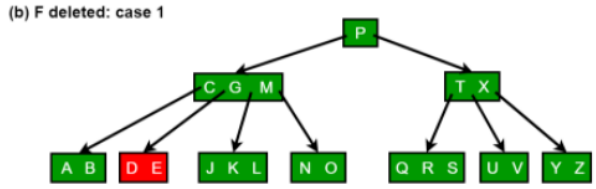
```
public void Insert(int key) {
    BNode r = root;
    if (r.n == 2*t-1) {
        BNode s = new BNode( n: 0,  leaf: false);
        root = s;
        s.child[0] = r;
        SplitChild(s,  pos: 0, r);
        InsertNonfull(s, key);
    } else {
        InsertNonfull(r, key);
    }
}
```

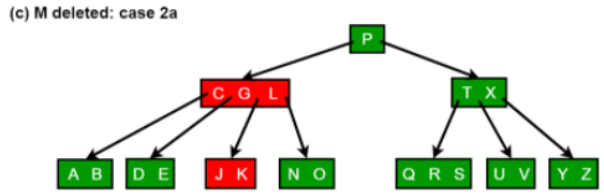
3.Delete:

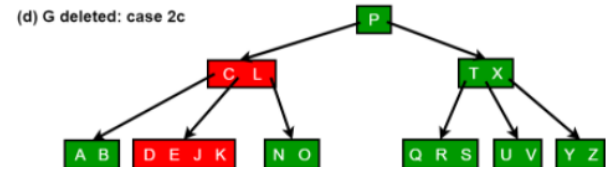
B-tree šalinimo metodas yra labai panašus į įterpimo metodą, tik labiau komplikuotas. Mes galime ištrinti bet kurį narį iš bet kurio ,,Node“, ir kai tai padarome mums teks medį perorganizuoti. Ir tuo pačiu įsitikinti, ar „Node“ nepasiekia minimumo (t-1).

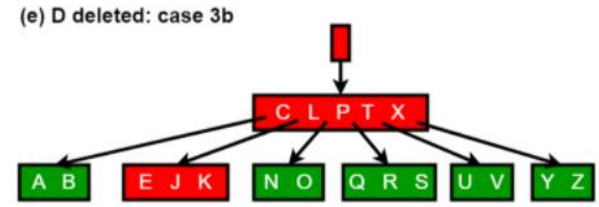
Vizualizacija:

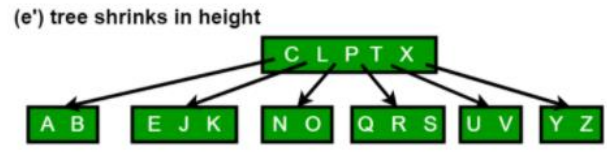


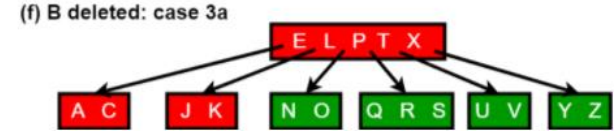












Mano kalboje realizacija:

1.B-tree delete:

```
public void delete(int key) {
    root.delete(key);

    if (!root.leaf && root.n == 0)
        root = root.child[0];
}
```

2.B-Node delete:

```
public void delete(int k) {
    if (leaf)
        deleteFromLeaf(k);
    else {
        int i = 0;

        while (i < n && keys[i] < k)
            i++;
        if (i < n && keys[i] == k)
            deleteFromInternalNode(i);
        else {

            BNode childTemp = child[i];
            ensureFullEnough(i);
            childTemp.delete(k);
        }
    }
}
```

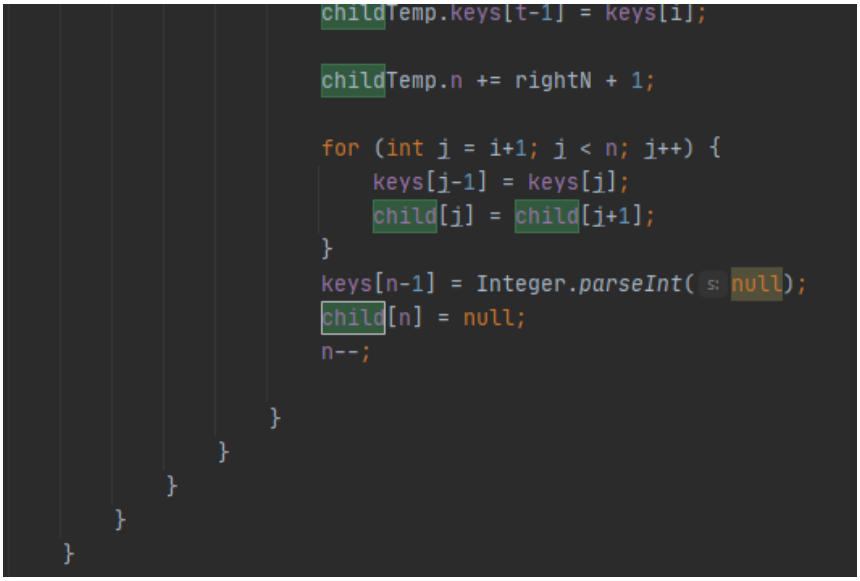
3.B-Node ensureFullEnough:

```
private void ensureFullEnough(int i) {
    BNode childTemp = child[i];
    if (childTemp.n < t) {
        BNode leftSibling;
        Integer leftN;

        if (i > 0) {
            leftSibling = child[i-1];
            leftN = leftSibling.n;
        }
        else {
            leftSibling = null;
            leftN = 0;
        }

        if (leftN >= t) {
            for (int j = childTemp.n-1; j >= 0; j--)
                childTemp.keys[j+1] = childTemp.keys[j];
            if (!childTemp.leaf)
                for (int j = childTemp.n; j >= 0; j--)
                    childTemp.child[j+1] = childTemp.child[j];

            // Move a key down from this node into child,from the left sibling into this node.
            childTemp.keys[0] = keys[i-1];
            keys[i-1] = leftSibling.keys[leftN-1];
            leftSibling.keys[leftN-1] = null;


            if (!childTemp.leaf) {
                childTemp.child[0] = leftSibling.child[leftN]; //  move a pointer from the left sibling into child
                leftSibling.child[leftN] = null;
            }
```

```
    leftSibling.n--;
    childTemp.n++;
}
else {      // Same at the right
    BNode rightSibling;
    Integer rightN;

    if (i < n) {
        rightSibling = child[i+1];

        rightN = rightSibling.n;
    }
    else {
        rightSibling = null;
        rightN = 0;
    }

    if (rightN >= t) {
        childTemp.keys[childTemp.n] = keys[i];
        keys[i] = rightSibling.keys[0];

        if (!childTemp.leaf)
            childTemp.child[childTemp.n] = rightSibling.child[0];

        for (int j = 1; j < rightN; j++)
            rightSibling.keys[j-1] = rightSibling.keys[j];
        rightSibling.keys[rightN-1] = null;
        if (!rightSibling.leaf) {
            for (int j = 1; j <= rightN; j++)
                rightSibling.child[j-1] = rightSibling.child[j];
            rightSibling.child[rightN] = null;
        }
```

```
    rightSibling.n--;
    childTemp.n++;

}
else {
    if (leftN > 0) {
        // Merge the left sibling into the child.
        // Start by moving everything in the child
        // right by t positions.
        for (int j = childTemp.n-1; j >= 0; j--)
            childTemp.keys[j+t] = childTemp.keys[j];
        if (!childTemp.leaf)
            for (int j = childTemp.n; j >= 0; j--)
                childTemp.child[j+t] = childTemp.child[j];

        // Take everything from the left sibling.
        for (int j = 0; j < leftN; j++) {
            childTemp.keys[j] = leftSibling.keys[j];
            leftSibling.keys[j] = Integer.parseInt( s: null);
        }
        if (!childTemp.leaf)
            for (int j = 0; j <= leftN; j++) {
                childTemp.child[j] = leftSibling.child[j];
                leftSibling.child[j] = null;
            }

        // Move a key down from this node into the child.
        childTemp.keys[t-1] = keys[i-1];

        childTemp.n += leftN + 1;

        for (int j = i; j < n; j++) {
```

```
        keys[j-1] = keys[j];
        child[j-1] = child[j];
    }
    child[n-1] = child[n];
    keys[n-1] = Integer.parseInt( s: null);
    child[n] = null;
    n--;

}
else {
    // Merge the right sibling into the child.
    for (int j = 0; j < rightN; j++) {
        childTemp.keys[j+childTemp.n+1] = rightSibling.keys[j];
        rightSibling.keys[j] = Integer.parseInt( s: null);
    }
    if (!childTemp.leaf)
        for (int j = 0; j <= rightN; j++) {
            childTemp.child[j+childTemp.n+1] = rightSibling.child[j];
            rightSibling.child[j] = null;
        }


    childTemp.keys[t-1] = keys[i];

    childTemp.n += rightN + 1;

    for (int j = i+1; j < n; j++) {
        keys[j-1] = keys[j];
        child[j] = child[j+1];
    }
    keys[n-1] = Integer.parseInt( s: null);
    child[n] = null;
```


childTemp.keys[t-1] = keys[i];
childTemp.n += rightN + 1;
for (int j = i+1; j < n; j++) {
keys[j-1] = keys[j];
child[j] = child[j+1];
}
keys[n-1] = Integer.parseInt(s: null);
child[n] = null;
n--;
}
}
}
}
}

4.Testavimas:

Kiekvienas metodas testuojamas pasinaudojant ,,Integer”, dėl patogumo ir aiškumo. Bet galima naudoti bet kokį duomenų tipą, nes yra įterptas ,,Interface“.

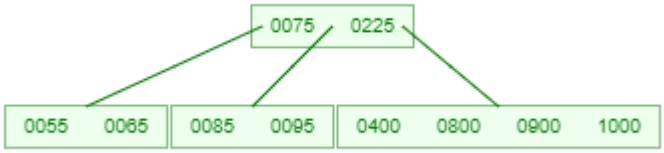
1.Search metodo testavimas:

Kodas:

```
public static void main(String[] args) {
    BTree b = new BTree( t: 3);
    b.Insert(new Element( g: 55));
    b.Insert(new Element( g: 65));
    b.Insert(new Element( g: 75));
    b.Insert(new Element( g: 85));
    b.Insert(new Element( g: 95));
    b.Insert(new Element( g: 225));
    b.Insert(new Element( g: 400));
    b.Insert(new Element( g: 800));
    b.Insert(new Element( g: 900));
    b.Insert(new Element( g: 1000));

    Object k = b.search(new Element( g: 800));


    b.display();

}
```

Rezultatai:

Vizualizacija:



2.Insert metodo testavimas:

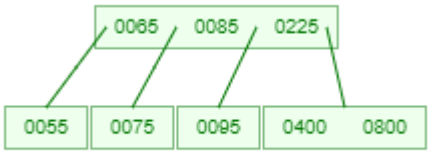
Kodas:

```
BTree b = new BTree( t: 2);
b.Insert(new Element( g: 55));
b.Insert(new Element( g: 65));
b.Insert(new Element( g: 75));
b.Insert(new Element( g: 85));
b.Insert(new Element( g: 95));
b.Insert(new Element( g: 225));
b.Insert(new Element( g: 400));
b.Insert(new Element( g: 800));
```

Rezultatai:

```
|65 85 225 |55 |75 |95 |400 800
```

Vizualizacija:



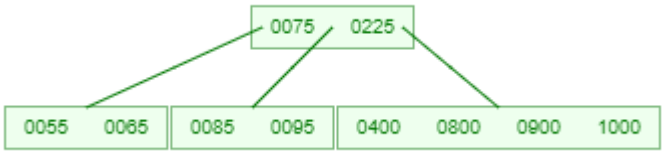
Kodas:

```
BTree b = new BTree( t: 3);
b.Insert(new Element( g: 55));
b.Insert(new Element( g: 65));
b.Insert(new Element( g: 75));
b.Insert(new Element( g: 85));
b.Insert(new Element( g: 95));
b.Insert(new Element( g: 225));
b.Insert(new Element( g: 400));
b.Insert(new Element( g: 800));
b.Insert(new Element( g: 900));
b.Insert(new Element( g: 1000));



b.display();
```

Rezultatai:

```
|75 225 |55 65 |85 95 |400 800 900 1000
```

Vizualizacija:



3.Delete metodo testavimas:

Kodas:

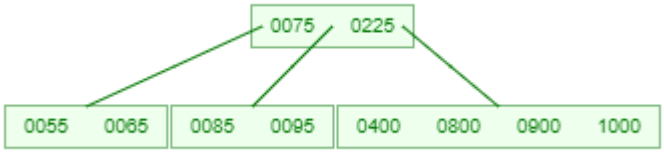
```
BTree b = new BTree( t: 3);
b.Insert(new Element( g: 55));
b.Insert(new Element( g: 65));
b.Insert(new Element( g: 75));
b.Insert(new Element( g: 85));
b.Insert(new Element( g: 95));
b.Insert(new Element( g: 225));
b.Insert(new Element( g: 400));
b.Insert(new Element( g: 800));
b.Insert(new Element( g: 900));
b.Insert(new Element( g: 1000));

b.delete(new Element( g: 400));


b.display();
```

Rezultatai:

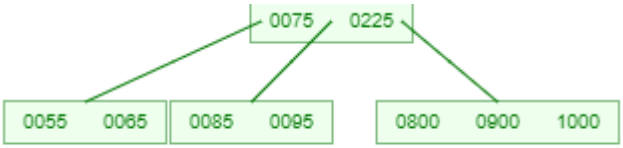
```
|225 75 |1000 900 800 |95 85 |65 55
```

Vizualizacija:

Prieš:



Po:



Kodas:

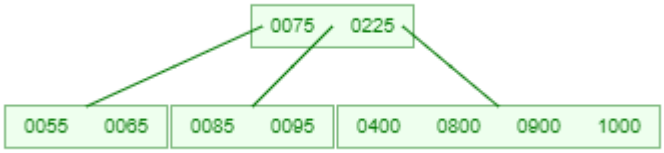
```
public static void main(String[] args) {
    BTree b = new BTree( t: 3);
    b.Insert(new Element( g: 55));
    b.Insert(new Element( g: 65));
    b.Insert(new Element( g: 75));
    b.Insert(new Element( g: 85));
    b.Insert(new Element( g: 95));
    b.Insert(new Element( g: 225));
    b.Insert(new Element( g: 400));
    b.Insert(new Element( g: 800));
    b.Insert(new Element( g: 900));
    b.Insert(new Element( g: 1000));

    b.delete(new Element( g: 225));


    b.display();
```

Rezultatai:

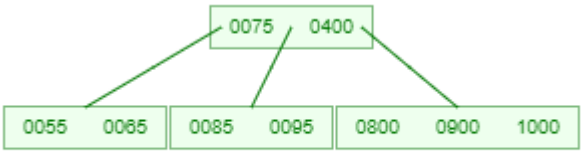
```
|400 75 |1000 900 800 |95 85 |65 55
```

Vizualizacija:

Prieš:

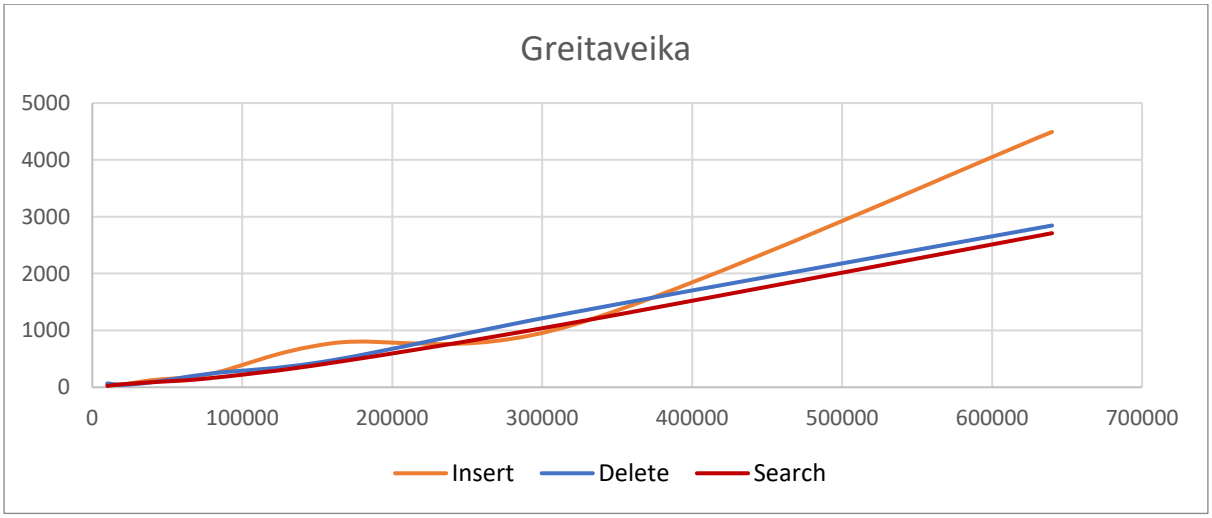


Po:



5.Greitaveika:

Šioje skiltyje atlikau greitaveikos testavimą. Testavau visus pagrindinius B-Tree metodus (realizacijoje „minimum degree“ yra 60).



Noriu pabrėžti, kad laikas matuojamas ms. Laikas yra y ašis, o kiekis yra x ašis. Tai pat verta paminėti, kad tyrimas buvo atliktas porą kartų ir duomenys beveik nesikeitė.

6.Išvados

Šiame technologiniame projekte man pavyko realizuoti duomenų struktūrą B-Tree bei pagrindinius jos metodus „Insert“, „Search“, „Delete“. Testavimo rezultatai parodo, kad metodai („Insert“, „Search“, „Delete“) veikia korektiškai, tačiau greitaveikos rezultatai parodo, kad tarp teorinio algoritmų sudėtingumo ir mano gauto praktinio algoritmų sudėtingumo yra skirtumų.

Teorinis:

1. „Search“, algoritmo sudėtingumas O(logn)
2. „Insert“, algoritmo sudėtingumas O(logn)
3. „Delete“, algoritmo sudėtingumas O(logn)

Mano gauti rezultatai:

1. „Search“, algoritmo sudėtingumas O(n)
2. „Insert“, algoritmo sudėtingumas O(nlogn)
3. „Delete“, algoritmo sudėtingumas O(n)

Mano gautuose rezultatuose „Search“ ir „Delete“ turi labai panašią greitaveiką. Abu yra tiesiškai priklausomi nuo duomenų kiekio ir tai prieštarauja teoriniam „Search“ ir „Delete“ logaritminiam sudėtingumui. Tai pat „Insert“ metodo atvaizdavime matome svyravimų. Jo atsiradimui įtakos turėjo tai, kad medyje nebebuvo laisvų raktų ir ši duomenų struktūra turėjo pasitelkti papildomą metodą „split child“ (šis metodas pavaizduotas „Insert“ metodo skiltyje). Šis metodas padeda sukurti papildomos vietos, bet tuo pačiu perskirsto medį. Ir dėl šios priežasties grafe galėjo atsirasti svyravimas.