Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

# Курсовая работа

Дисциплина: Программирование на языках высшего уровня

Тема: B - дерево

Выполнил студент гр. 3331506/70401 Ляховский М. В.

Преподаватель Ананьевский М. С.

«____» _____ 2020 г.

Санкт-Петербург
2020

Оглавление

## Введение

B-дерево - структура данных, дерево поиска. С точки зрения внешнего логического представления, сбалансированное, сильно ветвистое дерево. Часто используется для хранения данных во внешней памяти. Сбалансированность означает, что длина любых двух путей от корня до листьев различается не более, чем на единицу. Ветвистость дерева — это свойство каждого узла дерева ссылаться на большое число узлов-потомков. С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц памяти, то есть каждому узлу дерева соответствует блок памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

## Применение

Структура B-дерева применяется для организации индексов во многих современных СУБД. B-дерево может применяться для структурирования (индексирования) информации на жёстком диске (как правило, метаданных). Время доступа к произвольному блоку на жёстком диске очень велико (порядка миллисекунд), поскольку оно определяется скоростью вращения диска и перемещения головок. Поэтому важно уменьшить количество узлов, просматриваемых при каждой операции. Использование поиска по списку каждый раз для нахождения случайного блока могло бы привести к чрезмерному количеству обращений к диску вследствие необходимости последовательного прохода по всем его элементам, предшествующим заданному, тогда как поиск в B-дереве, благодаря свойствам сбалансированности и высокой ветвистости, позволяет значительно сократить количество таких операций. Относительно простая реализация алгоритмов и существование готовых библиотек (в том числе для C) для работы со структурой B-дерева обеспечивают популярность применения такой организации памяти в самых разнообразных программах, работающих с большими объёмами данных.

## Структура и принципы построения

B-деревом называется дерево, удовлетворяющее следующим свойствам:

1. Ключи в каждом узле обычно упорядочены для быстрого доступа к ним. Корень содержит от 1 до 2t-1 ключей. Любой другой узел содержит от t-1 до 2t-1 ключей. Листья не являются исключением из этого правила. Здесь t — параметр дерева, не меньший 2.
2. У листьев потомков нет. Любой другой узел, содержащий ключи $K_1, \dots, K_n$ содержит $n + 1$ потомков. При этом
    - Первый потомок и все его потомки содержат ключи из интервала $(-\infty, K_1)$

- Для $2 \leq i \leq n, i-$ый потомок и все его потомки содержат ключи из интервала $(K_{i-1}, K_i)$
- $(n+1)-$ый потомок и все его потомки содержат ключи из интервала $(K_n, \infty)$

3. Глубина всех листьев одинакова.

## Поиск

Если ключ содержится в корне, он найден. Иначе определяем интервал и идём к соответствующему потомку. Повторяем.

## Добавление ключа

Будем называть деревом потомков некоего узла поддерево, состоящее из этого узла и его потомков.

Вначале определим функцию, которая добавляет ключ $K$ к дереву потомков узла $x$. После выполнения функции во всех пройденных узлах, кроме, может быть, самого узла $x$, будет меньше $2t-1$, но не меньше $t-1$, ключей.

1. Если $x$ — не лист,
   - Определяем интервал, где должен находиться $K$. Пусть y — соответствующий потомок.
   - Рекурсивно добавляем $K$ к дереву потомков $y$.
   - Если узел y полон, то есть содержит $2t-1$ ключей, расщепляем его на два. Узел $y_1$ получает первые $t-1$ из ключей $y$ и первые t его потомков, а узел — $y_2$ последние $t-1$ из ключей $y$ и последние его потомков. Медианный из ключей узла $y$ попадает в узел $x$, а указатель на $y$ в узле $x$ заменяется указателями на узлы $y_1$ и $y_2$.
2. Если $x$ — лист, просто добавляем туда ключ $K$.

Теперь определим добавление ключа $K$ ко всему дереву. Буквой $R$ обозначается корневой узел.

1. Добавим $K$ к дереву потомков $R$
2. Если $R$ содержит теперь $2t-1$ ключей, расщепляем его на два. Узел $R_1$ получает первые $t-1$ из ключей $R$ и первые $t$ его потомков, а узел — $R_2$ последние $t-1$ из ключей $R$ и последние $t$ его потомков. Медианный из ключей узла $R$ попадает во вновь созданный узел, который становится корневым. Узлы $R_1$ и $R_2$ становятся его потомками.

## Удаление ключа

Если корень одновременно является листом, то есть в дереве всего один узел, мы просто удаляем ключ из этого узла. В противном случае сначала находим узел, содержащий ключ, запоминая путь к нему. Пусть этот узел — $x$.

Если $x$ — лист, удаляем оттуда ключ. Если в узле $x$ осталось не меньше $t - 1$ ключей, мы на этом останавливаемся. Иначе мы смотрим на количество ключей в следующем, а потом в предыдущем узле. Если следующий узел есть, и в нём не менее $t$ ключей, мы добавляем в $x$ ключ-разделитель между ним и следующим узлом, а на его место ставим первый ключ следующего узла, после чего останавливаемся. Если это не так, но есть предыдущий узел, и в нём не менее $t$ ключей, мы добавляем в $x$ ключ-разделитель между ним и предыдущим узлом, а на его место ставим последний ключ предыдущего узла, после чего останавливаемся. Наконец, если и с предыдущим ключом не получилось, мы объединяем узел $x$ со следующим или предыдущим узлом, и в объединённый узел перемещаем ключ, разделяющий два узла. При этом в родительском узле может остаться только $t - 2$ ключей. Тогда, если это не корень, мы выполняем аналогичную процедуру с ним. Если мы в результате дошли до корня, и в нём осталось от 1 до $t - 1$ ключей, делать ничего не надо, потому что корень может иметь и меньше $t - 1$ ключей. Если же в корне не осталось ни одного ключа, исключаем корневой узел, а его единственный потомок делаем новым корнем дерева.

Если $x$ — не лист, а $K$ — его $i$-й ключ, удаляем самый правый ключ из поддерева потомков $i$-го потомка $x$, или, наоборот, самый левый ключ из поддерева потомков $i$+1-го потомка $x$. После этого заменяем ключ K удалённым ключом. Удаление ключа происходит так, как описано в предыдущем абзаце.

# Код алгоритма

```cpp
#include <iostream>

using namespace std;

/*
*  t - max pointers in one node
*  t-1 - max amount of data in one node
*/

class Node
{
public:
    int *DataArray;
    Node **AddressArray;
    Node *ParentAddress;

    //current amount of numbers in node counter
    int DataCounter;

    //node leaf status flag
    bool leaf;

    //create new node
    Node(int data, int t)
    {
        //create data array
        this->DataArray = new int[t-1];

        for (int i = 0; i < t - 1; i++)
        {
            this->DataArray[i] = NULL;
            //this->DataArray[i] = i;
        }

        //create address array
        this->AddressArray = new Node *[t];

        for (int i = 0; i < t; i++)
        {
            this->AddressArray[i] = nullptr;
        }

        //write new data to node array
        this->DataArray[0] = data;

        this->DataCounter = 1;

        ParentAddress = nullptr;

        this->leaf = true;
```

```cpp
        cout << "Node constructor" << endl;
    }

private:

};

class Tree
{
public:
    Node *root;

    Tree(int t)
    {
        root = nullptr;

        this->t = t;

        cout << "Tree constructor" << endl;
    }

    //add new data to the tree
    void AddData(int NewData);

    //find and write new data to appropriate place
    void FindAppropriatePlace(int NewData, Node *address);

    //mode elements in node data array forward from appropriate place
    void MoveElementsForward(Node *address, int AppropriatePlace);

    //mode elements in node data array backaward from appropriate place
    void MoveElementsBackward(Node* address, int AppropriatePlace);

    //split node into two parts, returns address of new created node with right half of splitted one
    Node *SplitNode(int NewData, Node *address);

    //find and delete goal data
    bool DeleteData(int DataToDelete, Node *address);

    void DeleteNode(Node *address);

    //insert new data to specific node
    void InsertData(int NewData, Node *address);

    //find data in specific node and retrn its place
    int FindDataInNode(int GoalData, Node *address);

    Node *FindData(int GoalData, Node* address);

    void PrintTree(Node *address);
```

```
    void PrintTree(Node *address);

private:
    unsigned int t;
};

void Tree::AddData(int NewData)
{
    //if tree has no root -> create root node
    if (root == nullptr)
    {
        root = new Node(NewData, this->t);
        return;
    }

    //find and write to appropriate place
    FindAppropriatePlace(NewData, root);
}

void Tree::FindAppropriatePlace(int NewData, Node *address)
{
    //all elements check
    for (int i = 0; i < (address->DataCounter); i++)
    {
        //if new data less than current element
        if (NewData < address->DataArray[i])
        {
            //and if this node is a leaf
            if (address->leaf == true)
            {
                //and if this node has empty space
                if (address->DataCounter < (t - 1))
                {
                    //move existing elements forward to set free appropriate place for new data
                    MoveElementsForward(address, i);

                    //write new data
                    address->DataArray[i] = NewData;
                    address->DataCounter++;

                    //exit
                    return;
                }
                //if this node has no empty space
                else
                {
                    //split node and write new data to appropriate place
                    SplitNode(NewData, address);
                    return;
                    //find new place and write new data to it
                    //return FindAppropriatePlace(NewData, root);
```

```
                    //return FindAppropriatePlace(NewData, root);
                }
            }
            //if this node is not a leaf
            else
            {
                //search deep
                return FindAppropriatePlace(NewData, address->AddressArray[i]);
            }
        }
    }

    //check is over, new data is more than all elements in node data array
    //hence we have to place new data to the right side of current node
    //check if it is a leaf
    if (address->leaf == true)
    {
        //if this node has empty space
        if (address->DataCounter < (t - 1))
        {
            //write new data
            address->DataArray[address->DataCounter] = NewData;
            address->DataCounter++;
        }
        else
        {
            //split node
            SplitNode(NewData, address);
            //return FindAppropriatePlace(NewData, root);
        }
    }
    //if it is not a leaf
    else
    {
        //search deep inlast child
        return FindAppropriatePlace(NewData, address->AddressArray[address->DataCounter]);
    }
}

void Tree::MoveElementsForward(Node *address, int AppropriatePlace)
{
    //index for moving left
    int index = address->DataCounter - 1;

    while (index >=0 && index != (AppropriatePlace - 1))
    {
        //copy current element forward
        address->DataArray[index + 1] = address->DataArray[index];
        address->AddressArray[index + 2] = address->AddressArray[index + 1];

        index--;
```

```
201         index--;
202       }
203
204       //after this action i can write new data to appropriate place in leaf node
205     }
206
207     void Tree::MoveElementsBackward(Node *address, int AppropriatePlace)
208     {
209       //index for moving right
210       int index = AppropriatePlace;
211
212       if (index == 0)
213       {
214         while (index < (address->DataCounter))
215         {
216           //copy current element backward
217           address->DataArray[index] = address->DataArray[index + 1];
218
219           index++;
220         }
221
222         index = 0;
223
224         while (index < (address->DataCounter + 1))
225         {
226           //copy current element backward
227           address->AddressArray[index] = address->AddressArray[index + 1];
228
229           index++;
230         }
231       }
232       else
233       {
234         while (index < (address->DataCounter))
235         {
236           //copy current element backward
237           address->DataArray[index] = address->DataArray[index + 1];
238           address->AddressArray[index + 1] = address->AddressArray[index + 2];
239
240           index++;
241         }
242       }
243
244       address->DataArray[address->DataCounter] = NULL;
245       address->AddressArray[address->DataCounter + 1] = nullptr;
246     }
247
248     Node *Tree::SplitNode(int NewData, Node *address){ ... }
478
479     void Tree::InsertData(int NewData, Node  *address){ ... }
```

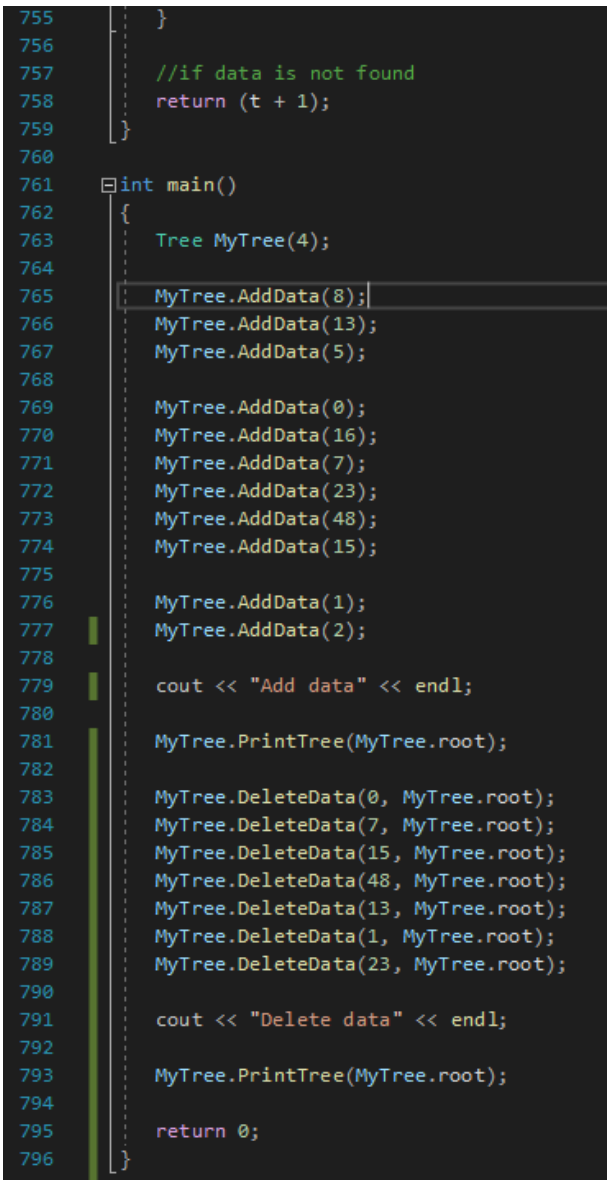
```
505
506     bool Tree::DeleteData(int DataToDelete, Node *address)
507     {
508       //find data to delete and return address of its node
509       Node *NodeAddress = FindData(DataToDelete, address);
510
511       //if data is not found
512       if (NodeAddress == nullptr)
513       {
514         return 0;
515       }
516       //if data to delete is found
517       else// if(NodeAddress->leaf == true)
518       {
519         //find place of deleting data
520         int PlaceToDelete = FindDataInNode(DataToDelete, NodeAddress);
521
522         //delete data
523         NodeAddress->DataArray[PlaceToDelete] = NULL;
524         NodeAddress->DataCounter--;
525
526         //move data backwards
527         MoveElementsBackward(NodeAddress, PlaceToDelete);
528
529         //if node has no data left
530         if (NodeAddress->DataCounter == 0)
531         {
532           int ChildAddressIndex = 0;
533
534           //find place in address array of node with deleted element in parent node
535           for (int i = 0; i <= NodeAddress->ParentAddress->DataCounter; i++)
536           {
537             //if address is equal
538             if (NodeAddress == NodeAddress->ParentAddress->AddressArray[i])
539             {
540               //write its index
541               ChildAddressIndex = i;
542
543               //break from for cycle
544               break;
545             }
546           }
547
548           Node *LeftNode = nullptr;
549           Node *RightNode = nullptr;
550           Node *ParentNode = NodeAddress->ParentAddress;
551
552           //if node with deleting data is first child of its parent
553           if (ChildAddressIndex == 0)
554           {
555             RightNode = NodeAddress->ParentAddress->AddressArray[ChildAddressIndex + 1];
```

```
555             RightNode = NodeAddress->ParentAddress->AddressArray[ChildAddressIndex + 1];
556           }
557           //or if node with deleting data is last child of its parent
558           else if (ChildAddressIndex == ParentNode->DataCounter + 1)
559           {
560             LeftNode = NodeAddress->ParentAddress->AddressArray[ChildAddressIndex - 1];
561           }
562           else
563           {
564             LeftNode = NodeAddress->ParentAddress->AddressArray[ChildAddressIndex - 1];
565             RightNode = NodeAddress->ParentAddress->AddressArray[ChildAddressIndex + 1];
566           }
567
568           //look left
569           //if there is left node and it has more than 1 element -> move data from parent node down
570           //and move last element of left node up to parent
571           if (LeftNode != nullptr && NodeAddress->ParentAddress->AddressArray[ChildAddressIndex - 1]->DataCounter > 1)
572           {
573             //move parent left data down
574             NodeAddress->DataArray[0] = ParentNode->DataArray[ChildAddressIndex - 1];
575             NodeAddress->DataCounter++;
576
577             //move biggest data from left node up
578             ParentNode->DataArray[ChildAddressIndex - 1] = LeftNode->DataArray[LeftNode->DataCounter - 1];
579             LeftNode->DataArray[LeftNode->DataCounter - 1] = NULL;
580             LeftNode->DataCounter--;
581           }
582           //or look right
583           //if there is right node and it has more than 1 element -> move data from parent node down
584           //and move last element of left node up to parent
585           else if(RightNode != nullptr && NodeAddress->ParentAddress->AddressArray[ChildAddressIndex + 1]->DataCounter > 1)
586           {
587             //move parent right data down
588             NodeAddress->DataArray[0] = ParentNode->DataArray[ChildAddressIndex];
589             NodeAddress->DataCounter++;
590
591             //move smallest data from right node up
592             ParentNode->DataArray[ChildAddressIndex] = RightNode->DataArray[0];
593             RightNode->DataArray[0] = NULL;
594             RightNode->DataCounter--;
595
596             //move elements of right node backward
597             MoveElementsBackward(RightNode, 0);
598           }
599           //if left and right node has 1 element
600           else
601           {
602
603             //look up
604             //if parent has nore than 1 element
605             if (ParentNode->DataCounter > 1)
```

```
605             if (ParentNode->DataCounter > 1)
606             {
607               //if left node exists
608               if (LeftNode != nullptr)
609               {
610                 //move left parent data of child address index to the right border of left node
611                 LeftNode->DataArray[LeftNode->DataCounter] = ParentNode->DataArray[ChildAddressIndex - 1];
612                 LeftNode->DataCounter++;
613                 ParentNode->DataCounter--;
614
615                 MoveElementsBackward(ParentNode, ChildAddressIndex - 1);
616
617                 DeleteNode(NodeAddress);
618               }
619               //if left node does not exists
620               else if (RightNode != nullptr)
621               {
622                 MoveElementsForward(RightNode, 0);
623
624                 //move first parents data element down to right node
625                 RightNode->DataArray[0] = ParentNode->DataArray[0];
626                 RightNode->DataCounter++;
627                 ParentNode->DataArray[0] = NULL;
628                 ParentNode->DataCounter--;
629
630                 MoveElementsBackward(ParentNode, 0);
631
632                 DeleteNode(NodeAddress);
633               }
634             }
635           }
636         }
637       }
638
639       //mission complete
640       return 1;
641     }
642
643     Node *Tree::FindData(int GoalData, Node *address)
644     {
645       //search for goal data
646       int DataPlace = FindDataInNode(GoalData, address);
647       Node *ResultNode;
648
649       //if data is not found
650       if (DataPlace == (t + 1))
651       {
652         //if address is root and a leaf
653         if (address == root && address->leaf == true)
654         {
655           //return nullptr (nothing found)
```

```
655             //return nullptr (nothing found)
656             return nullptr;
657           }
658           else
659           {
660             //compare all elements and find next appropriate node
661             //all elements check
662             for (int i = 0; i < (address->DataCounter); i++)
663             {
664               //if goal data less than current element
665               if (GoalData < address->DataArray[i])
666               {
667                 //if child address exists
668                 if (address->AddressArray[address->DataCounter] != nullptr)
669                 {
670                   //search deep
671                   ResultNode = FindData(GoalData, address->AddressArray[i]);
672
673                   if (ResultNode != nullptr)
674                   {
675                     return ResultNode;
676                   }
677                 }
678                 else
679                 {
680                   //return nullptr (nothing found)
681                   return nullptr;
682                 }
683               }
684             }
685
686             //goal data is more than existing data of current node
687             //if child address exists
688             if (address->AddressArray[address->DataCounter] != nullptr)
689             {
690               //move to right child
691               ResultNode = FindData(GoalData, address->AddressArray[address->DataCounter]);
692
693               if (ResultNode != nullptr)
694               {
695                 return ResultNode;
696               }
697             }
698             else
699             {
700               //return nullptr (nothing found)
701               return nullptr;
702             }
703           }
704         }
705         //if data is found
```

```
705         //if data is found
706         else
707         {
708           return address;
709         }
710       }
711
712       void Tree::DeleteNode(Node *address)
713       {
714         //if we need to delete root
715         if (address == root)
716         {
717           delete root;
718         }
719         //if we need to delete node
720         else
721         {
722           delete address;
723         }
724       }
725
726       void Tree::PrintTree(Node *address)
727       {
728         for (int i = 0; i < (t - 1); i++)
729         {
730           cout << address->DataArray[i] << ", ";
731         }
732
733         cout << endl;
734
735         for (int j = 0; j < t; j++)
736         {
737           if (address->AddressArray[j] != nullptr)
738           {
739             PrintTree(address->AddressArray[j]);
740           }
741         }
742
743       }
744
745       int Tree::FindDataInNode(int GoalData, Node* address)
746       {
747         //all elements check
748         for (int i = 0; i < (address->DataCounter); i++)
749         {
750           //if data equals current data element
751           if (GoalData == address->DataArray[i])
752           {
753             return i;
754           }
755         }
```



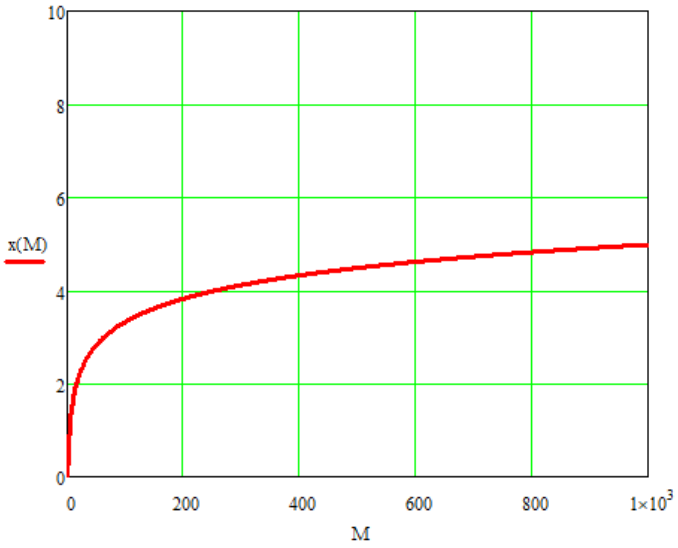
```
755         }
756
757         //if data is not found
758         return (t + 1);
759     }
760
761     int main()
762     {
763         Tree MyTree(4);
764
765         MyTree.AddData(8);
766         MyTree.AddData(13);
767         MyTree.AddData(5);
768
769         MyTree.AddData(0);
770         MyTree.AddData(16);
771         MyTree.AddData(7);
772         MyTree.AddData(23);
773         MyTree.AddData(48);
774         MyTree.AddData(15);
775
776         MyTree.AddData(1);
777         MyTree.AddData(2);
778
779         cout << "Add data" << endl;
780
781         MyTree.PrintTree(MyTree.root);
782
783         MyTree.DeleteData(0, MyTree.root);
784         MyTree.DeleteData(7, MyTree.root);
785         MyTree.DeleteData(15, MyTree.root);
786         MyTree.DeleteData(48, MyTree.root);
787         MyTree.DeleteData(13, MyTree.root);
788         MyTree.DeleteData(1, MyTree.root);
789         MyTree.DeleteData(23, MyTree.root);
790
791         cout << "Delete data" << endl;
792
793         MyTree.PrintTree(MyTree.root);
794
795         return 0;
796     }
```

## Анализ алгоритма

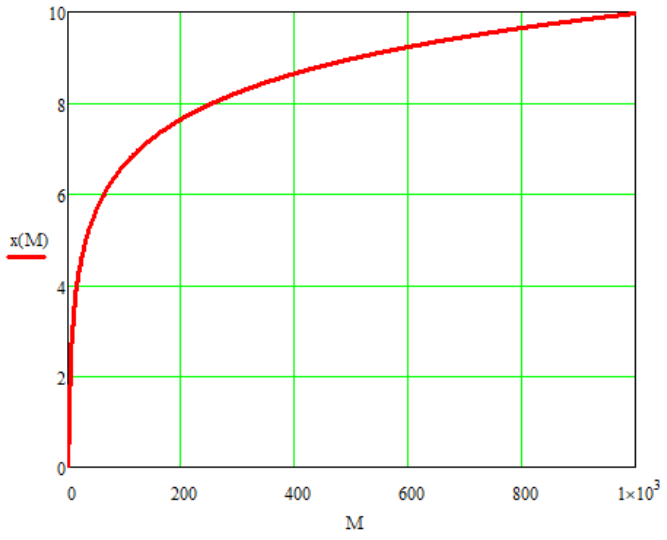
B - деревья сортированы: ключи внутри самого узла B- дерева находятся в порядке сортировки. Благодаря этому, для определения места некого искомого ключа мы можем применять некий алгоритм, подобный двоичному поиску. Это также подразумевает, что поиск по B - деревьям обладает логарифмической сложностью. Например, поиск искомого ключа среди 4 миллиардов ($4 \times 10^9$) элементов занимает около 32 сравнений.

Сложность поиска в B - дереве может быть представлена с двух точек зрения: общего числа блочного обмена и общего числа сравнений, осуществляемых в процессе такого поиска.

В терминах числа обменов, значением основания логарифма выступает $N$ (числа ключей в узле). На каждом новом уровне имеется в $K$ раз больше узлов, а следуя за неким дочерним указателем снижает пространство поиска на значение множителя $N$. В процессе поиска по крайней мере $\log_K M$ (где $M$ это общее число элементов в этом B - дереве) страниц для решения поиска искомого ключа. Общее число дочерних указателей, которым придётся следовать при проходе от вершины к листьям также равно общему числу уровней, иными словами, значению высоты $h$ данного дерева.




С точки зрения числа сравнений, значением основания логарифма является 2, так как поиск ключа внутри каждого узла осуществляется при помощи двоичного поиска. Каждое сравнение делит пополам пространство поиска, поэтому сложность равна $\log_2 M$.



## СПИСОК ЛИТЕРАТУРЫ


1. Левитин А. В. Глава 7. Пространственно-временной компромисс: B-деревья // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.
2. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Глава 18. B-деревья // Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е изд. — М.: Вильямс, 2006.