# Advanced Data Structures and Algorithm Analysis

## Laboratory Project2

# Self-printable B+ Tree

胡亮泽、徐雨豪、刘俊灏、毛晨炀

**Date: 2013-12-3**

# contents

**Chapter 1: Introduction .....3**

**Chapter 2: Data Structure / Algorithm Specification .....3**

**Chapter 3: Testing Results .....10**

**Chapter 4: Analysis and Comments .....13**

**Appendix: Source Code (if required) ....14**

**References.....** 错误！未定义书签。

**Author List .....21**

**Declaration .....21**

**Signatures .....** 错误！未定义书签。

## Chapter 1: Introduction

In this project , we are required to implement a B tree of order 3 with operations initialize, insert(with splitting) , and search. Of course, according to the name of our project , the B tree must can print out itself.

## Chapter 2: Data Structure / Algorithm Specification

### 2.1Data Structure

**This is the code of B tree structure:**

```
struct bTreenode
{
  int keynum; /*store keynum Data*/
  int data[M+1];    /*store each Data of the node*/
  btree child[M+1];
  btree parent;
  int is_leaf; /*1 represents it is a leaf,0 represents it isn't*/
};
```

Keynum stores the number of data stored in this node. The data array stores the keys and the child array stores the first key of each child between 2 and keynum.

The variable parent, obviously, stores the parent of the node ,which can be of much use later.

The variable is_leaf can be used to determine whether the node is a leaf or not.

### 2.2Algorithm Specification

Now we will show you our key algorithm(pseudocode).

#### 2.2.1Algorithm Specification of Initialize

This function is used to initialize a tree.

```
btree initialize(btree T)
{
    T = create_newnode();
```

```
    if(T==NULL) {
        printf("Out of Space!");
        return NULL;
    }
    return T;
}
```

### 2.2.2Algorithm Specification of Node Creation

This function is used to create a new node.

```
Node create_newnode()
{
    int i;
    Node newnode=NULL;
    newnode = (Node)malloc(sizeof(struct bTreenode));
    if(malloc failed) {
        printf("Out of Space!");
        return NULL;
    }

    for(i=0 to M)          /*initialize the data*/
        newnode->data[i] = -1;

    for(i =0 to M)              /*initialize the child*/
        newnode->child[i] = NULL;

    newnode->keynum = 0;
    newnode->is_leaf = True;
    newnode->parent = NULL;

    return newnode;
```

### 2.2.3Algorithm Specification of Search

This function is used to search a key and determine whether it is in the tree or not.

```
int search(int X,btree T) /*1 represents X is in the btree,0 represents it isn't*/
{
    int i;
```

```
    if(T is a leaf)
    {
        for(i=0 to keynum)
        {
            if(X==T->data[i])   /*Find X*/
                return 1;
        }
        return 0;                       /*X is not in the leaf*/
    }
    if(T is not a leaf)
    {
        for(i=0 to keynum)
        {
            if(X==T->data[i])          /*Find X*/
                return 1;
            else if(X<T->data[i])            /*X is in the child*/
                return search(X,T->child[i]);
        }
        return search(X,T->child[i]);
    }
}
```

### 2.2.4Algorithm Specification of Insert

This function is used to insert a key into the tree and split it if necessary.

Finally we just return the new tree after insertion.

```
btree    insert ( int X,    btree T )
{
    int i,j;
    /*The case that X is in the btree*/
    if(search(X,T)==1)
    {
        printf("Key %d is duplicated\n",X);
        return T;
    }

    /*Search from root to leaf for X and find the proper leaf node*/
    while(T is not a leaf)
    {
        for(i=0 to keynum)
        {
```

```
                if(X<T->data[i])
                {
                    T=T->child[i];
                    break;
                }
            }
            if(i==T->keynum)
                T=T->child[i];
        }
        /*End-Searching*/

        /*Insert X in the leaf*/
        for(i=0 to keynum)
        {
            if(X<T->data[i])
            {
                for(j=keynum to i+1)
                    T->data[j]=T->data[j-1];
                T->data[i]=X;
                break;
            }
        }
        if(i==T->keynum)
            T->data[i]=X;
        T->keynum++;
        /*End-Inserting X*/

        /*while this node has M+1 keys or M+1 children*/
        while((T is a leaf and T has M+1 keys) or T is not a leaf and T has M keys)
        {
            if(T->parent==NULL)                    /*this node is the root.create a new root
with two children;*/
                {
                    Node new_root = create_newnode();
                    new_root->keynum=0;
                    new_root->child[0]=T;
                    new_root->is_leaf=False;
                    T->parent=new_root;
                }
            split(T,T->parent);
            if(T->parent!=NULL)
                T=T->parent;   /*check its parent*/
        }/*End-While*/
```

```
    /*Return the root*/
    while(T->parent!=NULL)
        T=T->parent;
    return T;
}
```

### 2.2.5Algorithm Specification of Split

This is the split function used to split a node if its keynum exceed the max keynum.

```
void split(btree ch,btree newparent) /*split the node into 2 nodes with 2 keys, respectively*/
/*ch is the old child ,new_child is the new child. newparent is the 2 children's parent*/
{
    int i;
    Node new_child = create_newnode();

    If ch is a leaf, then new_child is a leaf;

    int pos;                    /*Get the position where ch is in the newparent->child[]*/
    for(i=0 to newparent->keynum)
    {
        if(newparent->child[i]==ch)
        {
            pos=i;
            break;
        }
    }

    /*update the information when if is a leaf or isn't respectively*/
    if(new_child is a leaf)
    {
        /*Update the child*/
        new_child->parent=newparent;
        new_child->data[0]=ch->data[2];
        new_child->data[1]=ch->data[3];
        ch->keynum=(M+1)/2;
        new_child->keynum=(M+1)/2;

        /*update the parent*/
        for(i = newparent->keynum to pos +1)
            newparent->child[i+1] = newparent->child[i];
```

```
            newparent->child[pos+1] = new_child;

            for(i = newparent->keynum – 1 to pos)
                  newparent->data[i+1] = newparent->data[i];
            newparent->data[pos] = new_child->data[0];

            newparent->keynum += 1;
      }

      if(new_child is not a leaf)
      {
            /*update the child*/
            new_child->parent=newparent;
            new_child->data[0]=ch->data[M-1];
            ch->keynum=M/2;
            new_child->keynum=M/2;

            new_child->child[0]=ch->child[2];
            new_child->child[1]=ch->child[3];
            ch->child[2]->parent=new_child;
            ch->child[3]->parent=new_child;

            /*update the parent*/
            for(i = newparent->keynum to pos+1)
                  newparent->child[i+1] = newparent->child[i];
            newparent->child[pos+1] = new_child;

            for(i = newparent->keynum – 1 to pos)
                  newparent->data[i+1] = newparent->data[i];
            newparent->data[pos] = ch->data[M/2];

            newparent->keynum += 1;
      }
}
```

### 2.2.6Algorithm Specification of Print

This function is used to print the tree. We use a queue to implement the operation.

```
void    printout_levelorder(btree T)/*Use a queue to print the btree by levelorder*/
{
```

```
    Node queue[10000] = {NULL};
    Node node=NULL;
    int front = 0;
    int rear = 0;
    int i;
    int store_rear;       /*Useful for reprenting whether one level has all been printed out*/

    queue[rear++] = T;       /*Enqueue*/
    store_rear=rear;
    for each level
    while(the queue is not enpty) {
        node = queue[front++];    /*Dequeue*/

        printf("[");
        for(i = 0 to node->keynum-1)
            printf("%d,", node->data[i]);
        printf("%d", node->data[i]);
        printf("]");

        for( i = 0 to node->keynum)
            if( node->child[i]!=NULL)
                queue[rear++] = node->child[i];

        if(front==the end of the level)      /*One level has been all printed out*/
        {
            printf("\n");
            store_rear=rear;
        }
    }
}
```

## 2.3Algorithm Specification of Main

```
int main()
{
    btree T=NULL;
    int i;/*for the loop*/
    int N;
    int X;
```

```
    T= initialize(T);    /*Initialize the btree*/

    scanf("%d",&N);
    for(i=0 to N-1)
    {
        scanf("%d",&X);
        insert X;
    }

    Printout T;

    Make empty T;
    return 0;
    system("pause");
}
```

## Chapter 3: Testing Results

We design several special cases to test our routine , the specification is shown in the table below. During the testing stage, some input has given the output contradict to the expected outcome. After debugging and analysis, we corrected the problems.

| case | Input | output | expected | description | status |
|---|---|---|---|---|---|
| 1 | 1 2 3 4 5 6 7 8 9 | [5]<br>[3][7]<br>[1,2][3,4][5,6][7,8,9] | [5]<br>[3][7]<br>[1,2][3,4][5,6][7,8,9] | Insertion in increasing order with a little data. | Pass |

| | | | | | |
|---|---|---|---|---|---|
| 2 | 9 8 7 6 5 4 3 2 1 | [6]<br>[4][8]<br>[1,2,3][4,5][6,7][8,9] | [6]<br>[4][8]<br>[1,2,3][4,5][6,7][8,9] | Insertion in decreasing order with a little data in order to be compared with the above example | Pass |
| 3 | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | [9,17]<br>[5][13][21]<br>[3][7][11][15][19][23,25]<br>[1,2][3,4][5,6][7,8][9,10][11,12][13,14][15,16][17,18][19,20][21,22][23,24][25,26,27] | [9,17]<br>[5][13][21]<br>[3][7][11][15][19][23,25]<br>[1,2][3,4][5,6][7,8][9,10][11,12][13,14][15,16][17,18][19,20][21,22][23,24][25,26,27] | Insertion in increasing order with a little data, too. Examining the correctness of the code when data is a little larger . | Pass |
| 4 | 94 398 943 182 323 16 655 779 75 205 773 24 8 940 199 995 296 250 112 439 842 451 129 928 994 924 268 424 859 827 627 913 276 482 481 723 950 23 94 40 621 711 745 695 29 333 929 726 855 365 789 276 907 838 645 686 312 463 392 200 187 729 303 158 469 630 714 763 560 712 433 609 362 501 574 2 283 593 282 152 116 252 276 744 816 38 225 364 784 592 544 344 649 813 214 205 600 858 930 629 | Key 94 is duplicated<br>Key 276 is duplicated<br>Key 276 is duplicated<br>Key 205 is duplicated<br>Key 629 is duplicated<br>[398,655]<br>[182,268][574][779,859]<br>[75][205][312,362][482][621][723][816][928,943]<br>[23,29][112,152][199][250][29 | Key 94 is duplicated<br>Key 276 is duplicated<br>Key 276 is duplicated<br>Key 205 is duplicated<br>Key 629 is duplicated<br>[398,655]<br>[182,268][574][779,859]<br>[75][205][312,362][482][621][723][816][928,943]<br>[23,29][112,152][199][250][29 | Insertion with 100 random number which less than 1000 in random order. Examining the correctness when we insert repeated number and insert in random order. What’s more, we prove that the code is able to run correctly when we insert large data. | Pass |

| | | | | | |
|---|---|---|---|---|---|
| | | 6,296][333][365][439,463][560][593][630][695][726,745][789][842][913][930][994]<br>[2,8,16][23,24][29,38,40][75,94][112,116,129][152,158][182,187][199,200][205,214,225][250,252][268,276][282,283][296,303][312,323][333,344][362,364][365,392][398,424,433][439,451][463,469][481,482][501,560][544,574,592][593,600,609][621,627,629][630,645,649][655,686][695,711][712,714,723][726,729,744][745,763,773][779,78 | 6,296][333][365][439,463][560][593][630][695][726,745][789][842][913][930][994]<br>[2,8,16][23,24][29,38,40][75,94][112,116,129][152,158][182,187][199,200][205,214,225][250,252][268,276][282,283][296,303][312,323][333,344][362,364][365,392][398,424,433][439,451][463,469][481,482][501,560][544,574,592][593,600,609][621,627,629][630,645,649][655,686][695,711][712,714,723][726,729,744][745,763,773][779,78 | | |

| | | | | | |
|---|---|---|---|---|---|
| | | 4][789,81<br>3][816,83<br>8][842,85<br>5,858][82<br>7,859][90<br>7,913,924]<br>[928,929][<br>930,940][<br>943,950][<br>994,995] | 4][789,81<br>3][816,83<br>8][842,85<br>5,858][82<br>7,859][90<br>7,913,924]<br>[928,929][<br>930,940][<br>943,950][<br>994,995] | | |

## Chapter 4: Analysis and Comments

### Analysis

The depth of the B tree is at most $\log_{M/2} N$, and logM is required to find the key. So the search operation takes O(log N) time to perform.

Insertion operation need O(M) time to fix up all information of the node. All node in the path may need this work in worst-case, so the insertion operation takes O($M\log_M N$). The splitting opration need O(M) time to fix up the information of a node, in worst-case , all node in the path need to be splitted , so the time complexity is the same. Thus we conclude the time complexity of insertion with splitting is O($M\log_M N$).

Finally , we need to traverse each node of the tree to print out the data. For each node, we need O(M) time to traverse each subtree of the node. So the total key stored is O(M*N), so it will take O(M*N) time to perform print out operation.

The space complexity, of course , is O(s*N), where s represents the space a node needs.

## Comments

This time we implement a few operations of B tree.

Our routine contains a lot of arrays, some may cause a waste of space and thus lead to a large space complexity. For example, each node contains a array of 4 element and it's hard for all of them to be full. Using a list can solve this problem, however, consider the time complexity grows because of the time used in deletion and creation, using an array is a better choice.

When print out the tree ,we use an array to implement a queue for simplicity, and possible use of pointer can improve the function.

Of course the routine is not perfect, but we have tried to make it the best we can.

## Appendix: Source Code (if required)

```
#include <stdio.h>
#include <stdlib.h>

#define M 3 /*The oreder of the B tree*/
#define True 1
#define False 0

#ifndef _BTREE_H    /*Declartion of the B tree*/

struct bTreenode;
typedef struct bTreenode *btree;
typedef struct bTreenode *Node;

btree insert(int X,btree T);
int search(int X,btree T);
void    printout_levelorder(btree T);
btree initialize(btree T);
Node create_newnode();
void split(Node ch,Node newparent);
btree MakeEmpty(btree T);

#endif/*_BTREE_H*/

struct bTreenode
{
```

```
    int keynum; /*store keynum Data*/
    int data[M+1];    /*store each Data of the node*/
    btree child[M+1];
    btree parent;
    int is_leaf; /*1 represents it is a leaf,0 represents it isn't*/
};

int main()
{
    btree T=NULL;
    int i; /*for the loop*/
    int N;
    int X;

    T= initialize(T);    /*Initialize the btree*/

    scanf("%d",&N);
    for(i=0;i<N;i++)
    {
        scanf("%d",&X);
        T=insert(X,T);
    }

    printout_levelorder(T);

    T=MakeEmpty(T);
    return 0;
    system("pause");
}

btree initialize(btree T)
{
    T = create_newnode();
    if(T==NULL) {
        printf("Out of Space!");
        return NULL;
    }
    return T;
}

Node create_newnode()
{
    int i;
    Node newnode=NULL;
```

```
    newnode = (Node)malloc(sizeof(struct bTreenode));
    if(newnode==NULL) {
        printf("Out of Space!");
        return NULL;
    }

    for(i = 0; i <= M; i++)          /*initialize the data*/
        newnode->data[i] = -1;

    for(i = 0; i <=M; i++)           /*initialize the child*/
        newnode->child[i] = NULL;

    newnode->keynum = 0;
    newnode->is_leaf = True;
    newnode->parent = NULL;

    return newnode;
}


int search(int X,btree T) /*1 represents X is in the btree,0 represents it isn't*/
{
    int i;
    if(T->is_leaf==True)
    {
        for(i=0;i<T->keynum;i++)
        {
            if(X==T->data[i])   /*Find X*/
                return 1;
        }
        return 0;                       /*X is not in the leaf*/
    }
    if(T->is_leaf==False)
    {
        for(i=0;i<T->keynum;i++)
        {
            if(X==T->data[i])          /*Find X*/
                return 1;
            else if(X<T->data[i])           /*X is in the child*/
                return search(X,T->child[i]);
        }
        return search(X,T->child[i]);
    }
}
```

```
btree    insert ( int X,    btree T )
{
      int i,j;
      /*The case that X is in the btree*/
      if(search(X,T)==1)
      {
            printf("Key %d is duplicated\n",X);
            return T;
      }

      /*Search from root to leaf for X and find the proper leaf node*/
      while(T->is_leaf==False)
      {
            for(i=0;i<T->keynum;i++)
            {
                  if(X<T->data[i])
                  {
                        T=T->child[i];
                        break;
                  }
            }
            if(i==T->keynum)
                  T=T->child[i];
      }
      /*End-Searching*/

      /*Insert X in the leaf*/
      for(i=0;i<T->keynum;i++)
      {
            if(X<T->data[i])
            {
                  for(j=T->keynum;j>i;j--)
                        T->data[j]=T->data[j-1];
                  T->data[i]=X;
                  break;
            }
      }
      if(i==T->keynum)
            T->data[i]=X;
      T->keynum++;
      /*End-Inserting X*/

      /*while this node has M+1 keys or M+1 children*/
```

```
    while((T->is_leaf==True&&T->keynum==M+1)||T->is_leaf==False&&T->keynum==M)
    {
        if(T->parent==NULL)                    /*this node is the root.create a new root
with two children;*/
            {
                Node new_root = create_newnode();
                new_root->keynum=0;
                new_root->child[0]=T;
                new_root->is_leaf=False;
                T->parent=new_root;
            }
        split(T,T->parent);
        if(T->parent!=NULL)
            T=T->parent;   /*check its parent*/
    }/*End-While*/

    /*Return the root*/
    while(T->parent!=NULL)
        T=T->parent;
    return T;
}


void split(btree ch,btree newparent) /*split the node into 2 nodes with 2 keys, respectively*/
/*ch is the old child ,new_child is the new child. newparent is the 2 children's parent*/
{
    int i;
    Node new_child = create_newnode();

    new_child->is_leaf = ch->is_leaf;

    int pos;                /*Get the position where ch is in the newparent->child[]*/
    for(i=0;i<=newparent->keynum;i++)
    {
        if(newparent->child[i]==ch)
        {
            pos=i;
            break;
        }
    }

    /*update the information when if is a leaf or isn't respectively*/
    if(new_child->is_leaf==True)
    {
```

```
            /*Update the child*/
            new_child->parent=newparent;
            new_child->data[0]=ch->data[2];
            new_child->data[1]=ch->data[3];
            ch->keynum=(M+1)/2;
            new_child->keynum=(M+1)/2;

            /*update the parent*/
            for(i = newparent->keynum; i > pos; i--)
                  newparent->child[i+1] = newparent->child[i];
            newparent->child[pos+1] = new_child;

            for(i = newparent->keynum - 1; i >= pos; i--)
                  newparent->data[i+1] = newparent->data[i];
            newparent->data[pos] = new_child->data[0];

            newparent->keynum += 1;
      }

      if(new_child->is_leaf==False)
      {
            /*update the child*/
            new_child->parent=newparent;
            new_child->data[0]=ch->data[M-1];
            ch->keynum=M/2;
            new_child->keynum=M/2;

            new_child->child[0]=ch->child[2];
            new_child->child[1]=ch->child[3];
            ch->child[2]->parent=new_child;
            ch->child[3]->parent=new_child;

            /*update the parent*/
            for(i = newparent->keynum; i > pos; i--)
                  newparent->child[i+1] = newparent->child[i];
            newparent->child[pos+1] = new_child;

            for(i = newparent->keynum - 1; i >= pos; i--)
                  newparent->data[i+1] = newparent->data[i];
            newparent->data[pos] = ch->data[M/2];

            newparent->keynum += 1;
      }
}
```

```
void    printout_levelorder(btree T)/*Use a queue to print the btree by levelorder*/
{
        Node queue[10000] = {NULL};
        Node node=NULL;
        int front = 0;
        int rear = 0;
        int i;
        int store_rear;         /*Useful for reprenting whether one level has all been printed out*/

        queue[rear++] = T;          /*Enqueue*/
        store_rear=rear;

        while(front < rear) {
                node = queue[front++];    /*Dequeue*/

                printf("[");
                for(i = 0; i < node->keynum-1; i++)
                        printf("%d,", node->data[i]);
                printf("%d", node->data[i]);
                printf("]");

                for( i = 0; i <= node->keynum; i++)
                        if( node->child[i]!=NULL)
                                queue[rear++] = node->child[i];

                if(front==store_rear)      /*One level has been all printed out*/
                {
                        printf("\n");
                        store_rear=rear;
                }
        }
}

btree MakeEmpty(btree T)      /*Free the memory*/
{
        int i;
        if(T!=NULL)
        {
                for(i=0;i<=T->keynum;i++)
                {
                        T->child[i]=MakeEmpty(T->child[i]);
                }
                free(T);
```

```
    }
    return NULL;
}
```

## References

[1]Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, *China Machine Press*, 2010.

## Author List

**Programmer:Yuhao Xu**

**Tester: Junhao Liu & Chenyang Mao**

**Report Writer: Liangze Hu**

## Declaration

***We hereby declare that all the work done in this project titled***

***"Tries and AVL Trees" is of our independent effort as a group.***