Data Structures

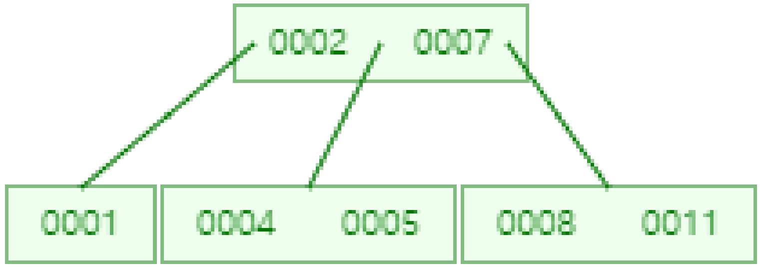
박영준 교수님

Lab9: B-Tree

B-Tree

- Self-balancing tree data structure that maintains sorted data
- B-tree of order(degree) M is a tree which satisfies the following properties:
  - 1. Every node has at most M child
  - 2. Every non-leaf node(except root) has at least $\lceil M/2 \rceil$ child
  - 3. Root has at least two child if it is not leaf
  - 4. Non-leaf node with K child contains $K-1$ keys
  - 5. All leaves appear in the same level and carry no information

B-Tree



B-Tree ADT

- NODE_DEGREE
  - Minimum # of childs a node must have
- MAX_CHILDS = NODE_DEGRE * 2
  - Maximum # of childs a node can have
- MAX_KEYS = MAX_CHILDS - 1
  - Maximum # of keys a node can have

B-Tree ADT

- typedef struct BTreeNode{

```
        int Keys[MAX_KEYS];
        struct BTreeNode *Childs[MAX_CHILDS];
        int KeyIndex;
        int Leaf;
};
```

HANYANG UNIVERSITY
DIVISION OF COMPUTER SCIENCE

B-Tree ADT

- BTreeNode *CreateBTreeNode();
  - Create BTreeNode and initiate it
  - Return new BTreeNode
- BTree CreateBTree();
  - Create BTree and initiate it
  - Return new BTree

B-Tree ADT

- void Insert(BTree *Tree, int Key);
  - Insert Key in Tree
  - If root is full, split root to make non full node and insert Key
  - If root is not full, insert Key in root

B-Tree ADT

- void InsertNonFull(BTreeNode *Node, int Key);
  - Find location to insert Key in non full node
  - If found child is full, split child to make non full node and insert Key

B-Tree ADT

- void SplitChild(BTreeNode *Parent, BTreeNode *Child, int Index);
  - Split Child
  - Separated childs has half of NODE_DEGREE keys
  - Link separated childs with Parent
  - Move key in median of child to parent

B-Tree ADT

- void Remove(BTree *Tree, int Key);
  - Remove Key in Tree
  - First, remove key
  - After remove, if root has no key
    - If root become leaf, there are no key in tree
    - Else, make child of root a new root
- int FindKey(BTreeNode *Node, int Key);
  - Find appropriate index of key and returns it

B-Tree ADT

- void RemoveNode(BTreeNode *Node, int Key);
  - Remove Key from Node or find child that is supposed to have Key
  - If Node has Key, remove it
  - If Node not has Key, find child that is supposed to have Key
  - If supposed child has less key than NODE_DEGREE, fill it
  - If Node not has Key and is Leaf, there are no Key in Tree

B-Tree ADT

- void RemoveFromLeaf(BTreeNode *Node, int Index);
  - Remove Key in leaf Node
- void RemoveFromNonLeaf(BTreeNode *Node, int Index);
  - Remove Key in non leaf Node
  - Another key must be replaced at the Key position
  - If both childs has less keys than NODE_DEGREE, need to merge

B-Tree ADT

- int RetPredecessor(BTreeNode *Node, int Index);
  - Find predecessor has maximum key
- int RetSuccessor(BTreeNode *Node, int Index);
  - Find successor has minimum key
- void Fill(BTreeNode *Node, int Index);
  - Fill Index-th child of Node with keys from other childs
  - If childs do not have enough key to lend, merge Index-th child with its sibling

B-Tree ADT

- void BorrowFromPrev(BTreeNode *Node, int Index);
  - Borrow key from previous child of Index-th child in Node
- void BorrowFromNext(BTreeNode *Node, int Index);
  - Borrow key from next child of Index-th child in Node
- void Merge(BTreeNode *Node, int Index);
  - Merge Index-th child of Node with its sibling

B-Tree ADT

- void Search(BTreeNode *Node, int Key);
  - Search Key
  - If Key is not found in Node and Node is Leaf, there are no Key in tree
  - If Key is not found in Node but Node is not Leaf, search recursively in the child

B-Tree

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

#define TRUE 1
#define FALSE 0

#define NODE_DEGREE      2 //Order
#define MAX_CHILDS   (NODE_DEGREE * 2)
#define MAX_KEYS          (MAX_CHILDS - 1)

#define SPACE (4 * MAX_KEYS)

typedef struct BTreeNode
{
    int Keys[MAX_KEYS];
    struct BTreeNode *Childs[MAX_CHILDS];
    int KeyIndex;
    int Leaf;
} BTreeNode;

typedef struct
{
    struct BTreeNode *Root;
    int Degree;
} BTree;

BTreeNode *CreateBTreeNode();
BTree *CreateBTree();

void Search( BTreeNode *Node, int Key);

void SplitChild(BTreeNode *Parent, BTreeNode *Child, int Index);

void InsertNonFull(BTreeNode *Node, int Key);
void Insert(BTree *Tree, int Key);

void Remove(BTree *Tree, int Key);

int FindKey(BTreeNode *Node, int Key);
void RemoveNode(BTreeNode *Node, int Key);

void RemoveFromLeaf(BTreeNode *Node, int Index);
void RemoveFromNonLeaf(BTreeNode *Node, int Index);

void Fill(BTreeNode *Node, int Index);

int RetPredecessor(BTreeNode *Node, int Index);
int RetSuccessor(BTreeNode *Node, int Index);

void BorrowFromPrev(BTreeNode *Node, int Index);
void BorrowFromNext(BTreeNode *Node, int Index);

void Merge(BTreeNode *Node, int Index);

void PrintTree(BTreeNode *Node, int Level, int Blank);
```

B-Tree

```c
void SplitChild(BTreeNode *Parent, BTreeNode *Child, int Index)
{
    //temporary stores Child
    BTreeNode *temp = CreateBTreeNode();

    temp->Leaf = Child->Leaf;
    temp->KeyIndex = NODE_DEGREE - 1;

    //move right half keys of Child to the temp
    for(int i = 0; i < NODE_DEGREE - 1; i++)
    {
        temp->Keys[i] = Child->Keys[NODE_DEGREE + i];
    }

    //if child is not leaf, move childs of child to the temp
    if(Child->Leaf == FALSE)
    {
        for(int i = 0; i < NODE_DEGREE; i++)
        {
            temp->Childs[i] = Child->Childs[NODE_DEGREE + i];
        }
    }
    Child->KeyIndex = NODE_DEGREE - 1;

    //move childs of parent, make space to insert temp
    for(int i = Parent->KeyIndex; i > Index - 1; i--)
    {
        Parent->Childs[i + 1] = Parent->Childs[i];
    }

    //insert temp
    Parent->Childs[Index] = temp;

    //move keys of parent, make space to insert keys of child
    for(int i = Parent->KeyIndex; i > Index - 1; i--)
    {
        Parent->Keys[i] = Parent->Keys[i - 1];
    }
    Parent->Keys[Index - 1] = Child->Keys[NODE_DEGREE - 1];

    Parent->KeyIndex++;
}
```

B-Tree

```c
void InsertNonFull(BTreeNode *Node, int Key)
{
    int Index = Node->KeyIndex;

    if(Node->Leaf == TRUE)
    {
        //find location to insert key
        while(Index > 0 && Node->Keys[Index - 1] > Key)
        {
            Node->Keys[Index] = Node->Keys[Index - 1];
            Index--;
        }
        Node->Keys[Index] = Key;
        Node->KeyIndex++;
    }
    else
    {
        //find location to insert key
        while(Index > 0 && Node->Keys[Index - 1] > Key)
        {
            Index--;
        }

        if(Node->Childs[Index]->KeyIndex == MAX_KEYS)
        {
            //if childs is full, split
            SplitChild(Node, Node->Childs[Index], Index + 1);

            //find child index to insert key
            if(Node->Keys[Index] < Key)
            {
                Index++;
            }
        }
        InsertNonFull(Node->Childs[Index], Key);
    }
}
```

B-Tree

```
void Insert(BTree *Tree, int Key)
{
    if(Tree->Root->KeyIndex == MAX_KEYS)
    {
        //if root is full, split
        BTreeNode *temp = CreateBTreeNode();

        temp->Leaf = FALSE;
        temp->KeyIndex = 0;
        temp->Childs[0] = Tree->Root;

        //make root is first child of new root
        SplitChild(temp, Tree->Root, 1);

        //insert key
        InsertNonFull(temp, Key);

        //change root pointer to new root
        Tree->Root = temp;
    }
    else
    {
        //if root is not full, insert key
        InsertNonFull(Tree->Root, Key);
    }
}
```

```
void Remove(BTree *Tree, int Key)
{
    //remove key first
    RemoveNode(Tree->Root, Key);

    if(Tree->Root->KeyIndex == 0)
    {
        //if root has no key
        if(Tree->Root->Leaf == TRUE)
        {
            //if root is leaf, there are no keys in tree
            Tree->Root =  CreateBTreeNode();
        }
        else
        {
            //else, change root to its child
            Tree->Root = Tree->Root->Childs[0];
        }
    }
}
```

```
int FindKey(BTreeNode *Node, int Key)
{
    //find key equal or greater than Key
    int Index = 0;
    while(Index < Node->KeyIndex && Node->Keys[Index] < Key)
    {
        Index++;
    }

    return Index;
}
```

B-Tree

```c
void RemoveNode(BTreeNode *Node, int Key)
{
    //find key index
    int Index = FindKey(Node, Key);

    //if key is exist in Node
    if(Index < Node->KeyIndex && Node->Keys[Index] == Key)
    {
        if(Node->Leaf == TRUE)
        {
            RemoveFromLeaf(Node, Index);
        }
        else
        {
            RemoveFromNonLeaf(Node, Index);
        }
    }
```

```c
    else
    {
        if(Node->Leaf == TRUE)
        {
            //if Node is leaf and has no key, there are no key in tree
            printf("Key %d is not exist in the tree\n", Key);
            return;
        }

        //indicator which node has key to be removed
        int Flag;
        if(Node->KeyIndex == Index)
        {
            //key in childs[index - 1]
            Flag = TRUE;
        }
        else
        {
            //key in childs[index]
            Flag = FALSE;
        }
```

B-Tree

```
        if(Node->Childs[Index]->KeyIndex < NODE_DEGREE)
        {
            //if the child where the key is supposed to exist has less key than NODE_DEGREE
            //need to fill the child
            Fill(Node, Index);
        }

        if(Flag && (Index > Node->KeyIndex))
        {
            //Key supposed in the childs[index - 1]
            RemoveNode(Node->Childs[Index - 1], Key);
        }
        else
        {
            //Key supposed in the childs[index]
            RemoveNode(Node->Childs[Index], Key);
        }
    }
}
```

B-Tree

```
void RemoveFromLeaf(BTreeNode *Node, int Index)
{
    //remove the index th key from node
    for(int i = Index + 1; i < Node->KeyIndex; i++)
    {
        Node->Keys[i - 1] = Node->Keys[i];
    }

    Node->KeyIndex--;
}
```

B-Tr

```
void RemoveFromNonLeaf(BTreeNode *Node, int Index)
{
    int Key = Node->Keys[Index];

    if(Node->Childs[Index]->KeyIndex > NODE_DEGREE - 1)
    {
        //if childs[index] has at least NODE_DEGREE keys
        //find predecessor and replace Keys[index] by predecessor-->remove Key
        int Pred = RetPredecessor(Node, Index);
        Node->Keys[Index] = Pred;
        //remove predecessor in childs[index]
        RemoveNode(Node->Childs[Index], Pred);
    }
    else if (Node->Childs[Index + 1]->KeyIndex > NODE_DEGREE - 1)
    {
        //if childs[index] has keys less than NODE_DEGREE,
        //and if childs[index + 1] has at least NODE_DEGREE keys
        //need to replace Key with its successor
        int Succ = RetSuccessor(Node, Index);
        Node->Keys[Index] = Succ;
        //remove successor in childs[index + 1]
        RemoveNode(Node->Childs[Index + 1], Succ);
    }
    else
    {
        //if both childs[index] and childs[index + 1] has less than NODE_DEGREE keys,
        //need merge both childs
        Merge(Node, Index);
        //after merge, Key is in the childs[index], remove it
        RemoveNode(Node->Childs[Index], Key);
    }

}
```

B-Tree

```
int RetPredecessor(BTreeNode *Node, int Index)
{
    //find predecessor has maximum key
    BTreeNode *Curr = Node->Childs[Index];
    while(Curr->Leaf == FALSE)
    {
        Curr = Curr->Childs[Curr->KeyIndex];
    }

    return Curr->Keys[Curr->KeyIndex - 1];
}

int RetSuccessor(BTreeNode *Node,  int Index)
{
    //find successor has minimum key
    BTreeNode *Curr = Node->Childs[Index + 1];
    while(Curr->Leaf == FALSE)
    {
        Curr = Curr->Childs[0];
    }

    return Curr->Keys[0];
}
```

B-Tree

```
void Fill(BTreeNode *Node, int Index)
{
    if(Index != 0 && Node->Childs[Index - 1]->KeyIndex > NODE_DEGREE - 1)
    {
        //if the prev child has more than NODE_DEGREE - 1 keys, borrow key from that child
        BorrowFromPrev(Node, Index);
    }
    else if(Index != Node->KeyIndex && Node->Childs[Index + 1]->KeyIndex > NODE_DEGREE - 1)
    {
        //if the next child has more than NODE_DEGREE - 1 keys, borrow key from that child
        BorrowFromNext(Node, Index);
    }
    else
    {
        //merge with its sibling
        if(Index != Node->KeyIndex)
        {
            //if child is not the last child, merge it with its next sibling
            Merge(Node, Index);
        }
        else
        {
            //if child is the last child, merge it with its prev sibling
            Merge(Node, Index - 1);
        }
    }
}
```

B-Tre

```c
void BorrowFromPrev(BTreeNode *Node, int Index)
{
    BTreeNode *Child = Node->Childs[Index];
    BTreeNode *Sibling = Node->Childs[Index - 1];

    //moving all key in childs[index] one step ahead
    for(int i = Child->KeyIndex - 1; i > -1; i--)
    {
        Child->Keys[i + 1] = Child->Keys[i];
    }

    //if childs[index] is not leaf, move all its child pointers one step ahead
    if(Child->Leaf == FALSE)
    {
        for(int i = Child->KeyIndex; i > -1; i--)
        {
            Child->Childs[i + 1] = Child->Childs[i];
        }
    }
    Child->Keys[0] = Node->Keys[Index - 1];

    //moving last child of sibling as first child of childs[index]
    if(Child->Leaf == FALSE)
    {
        Child->Childs[0] = Sibling->Childs[Sibling->KeyIndex];
    }
    Node->Keys[Index - 1] = Sibling->Keys[Sibling->KeyIndex - 1];

    Child->KeyIndex++;
    Sibling->KeyIndex--;

}
```

B-Tree

```
void BorrowFromNext(BTreeNode *Node, int Index)
{
    BTreeNode *Child = Node->Childs[Index];
    BTreeNode *Sibling = Node->Childs[Index + 1];

    //keys[index] is inserted as the last key in childs[index]
    Child->Keys[Child->KeyIndex] = Node->Keys[Index];

    //first child of sibling is inserted as the last child into childs[index]
    if(Child->Leaf == FALSE)
    {
        Child->Childs[Child->KeyIndex + 1] = Sibling->Childs[0];
    }

    //the first key from sibling is inserted into keys[index]
    Node->Keys[Index] = Sibling->Keys[0];

    //moving all keys in sibling one step behind
    for(int i= 1; i < Sibling->KeyIndex; i++)
    {
        Sibling->Keys[i - 1] = Sibling->Keys[i];
    }

    //moving the child pointers one step behind
    if(Sibling->Leaf == FALSE)
    {
        for(int i = 1; i < Sibling->KeyIndex; i++)
        {
            Sibling->Childs[i - 1] = Sibling->Childs[i];
        }
    }

    Child->KeyIndex++;
    Sibling->KeyIndex--;
}
```

```
void Merge(BTreeNode *Node, int Index)
{
    BTreeNode *Child = Node->Childs[Index];
    BTreeNode *Sibling = Node->Childs[Index + 1];

    //moving key from the Node and insert it into NODE_DEGREE -1 th location of childs[index]
    Child->Keys[NODE_DEGREE - 1] = Node->Keys[Index];

    //copying the keys from childs[index + 1 ] to childs[index] at the end
    for(int i = 0; i < Sibling->KeyIndex; i++)
    {
        Child->Keys[i + NODE_DEGREE] = Sibling->Keys[i];
    }

    //copying the child pointers from childs[index + 1] to childs[index]
    if(Child->Leaf == FALSE)
    {
        for(int i = 0; i < Sibling->KeyIndex + 1; i++)
        {
            Child->Childs[i + NODE_DEGREE] = Sibling->Childs[i];
        }
    }

    //moving all keys after index in the Node one step before to fill the space created by moving keys[index] to childs[index]
    for(int i = Index + 1; i < Node->KeyIndex; i++)
    {
        Node->Keys[i - 1] = Node->Keys[i];
    }

    //moving the child pointers after index + 1 in the Node one step before
    for(int i = Index + 2; i < Node->KeyIndex + 1; i++)
    {
        Node->Childs[i - 1] = Node->Childs[i];
    }

    Child->KeyIndex += Sibling->KeyIndex + 1;
    Node->KeyIndex--;
}
```

B-Tree

```
BTreeNode *CreateBTreeNode()
{
    BTreeNode *temp = (BTreeNode*)malloc(sizeof(BTreeNode));

    for(int i = 0; i < MAX_KEYS; i++)
    {
        temp->Keys[i] = 0;
    }

    for(int i = 0; i < MAX_CHILDS; i++)
    {
        temp->Childs[i] = NULL;
    }

    temp->KeyIndex = 0;
    temp->Leaf = TRUE;

    return temp;
}

BTree *CreateBTree()
{
    BTree *Tree = (BTree*)malloc(sizeof(BTree));

    BTreeNode *temp = CreateBTreeNode();

    Tree->Degree = NODE_DEGREE;//may not used in this code
    Tree->Root = temp;

    return Tree;
}
```

B-Tree

```c
void PrintTree(BTreeNode *Node, int Level, int Blank)
{
    //print b-tree as tree format
    if(Node->KeyIndex == 0)
    {
        printf("Tree not exist\n");
        return;
    }

    if(Blank == TRUE)
    {
        for(int i = 0; i < Level; i++)
        {
            for(int j = 0; j < SPACE; j++)
            {
                printf(" ");
            }
            printf("|");
        }
    }

    if(Node->KeyIndex > 0)
    {
        for(int i = 0; i < (SPACE - 4 * Node->KeyIndex); i++)
        {
            printf(" ");
        }

        printf("[");
        for(int i = 0; i < Node->KeyIndex - 1; i++)
        {
            printf("%2d, ", Node->Keys[i]);
        }
        printf("%2d]|", Node->Keys[Node->KeyIndex - 1]);
    }
    else
    {
        printf("[ ]");
    }
```

B-Tree

```

    if(Node->Leaf == TRUE)
    {
        printf("\n");
        return;
    }
    else
    {
        for(int i = 0; i < Node->KeyIndex + 1; i++)
        {
            if(i != 0)
            {
                PrintTree(Node->Childs[i], Level + 1, TRUE);
            }
            else
            {
                PrintTree(Node->Childs[i], Level + 1, FALSE);
            }
        }
    }
}
```

B-Tree

```c
int main(int argc, char *argv[])
{

    //targets of insert
    int test[] = {1, 7, 2, 11, 4, 8, 13, 10, 5, 19, 3, 6, 9, 18, 23, 12, 14, 20, 21, 16, 26, 27, 22, 24, 25};

    BTree *Tree = CreateBTree();
    int i;

    for(i=0; i<sizeof(test)/sizeof(int); i++)
    {
        printf("Insert : %d\n", test[i]);
        Insert(Tree, test[i]);
        PrintTree(Tree->Root, 0, FALSE);
        printf("\n");
    }

    for(int i = 0; i < 3 * (SPACE + 1); i++)
    {
        printf("-");
    }
    printf("\n");

```

B-Tree

```c
//targets of delete
int testdel[] = {14, 3, 12, 27, 1, 10, 11, 25, 5, 9, 23, 20, 8, 4, 24, 1};

for(i = 0; i < sizeof(testdel)/sizeof(int); i++)
{
    printf("Remove : %d\n", testdel[i]);
    Remove(Tree, testdel[i]);
    PrintTree(Tree->Root, 0, FALSE);
    printf("\n");
}

for(int i = 0; i < 3 * (SPACE + 1); i++)
{
    printf("-");
}
printf("\n");
```

B-Tree

```
    //search keys
    int Target = 2;
    printf("Search %d\n", Target);
    Search(Tree->Root, Target);
    printf("\n");

    Target = 26;
    printf("Search %d\n", Target);
    Search(Tree->Root, Target);
    printf("\n");

    Target = 18;
    printf("Search %d\n", Target);
    Search(Tree->Root, Target);
    printf("\n");

    Target = 14;
    printf("Search %d\n", Target);
    Search(Tree->Root, Target);
    printf("\n");
    return 0;
}
```

Lab9: B-Tree

- Submit on GitLab
- Complete Search in given code
- Create Lab9 directory on your own GitLab project
- Submit file : source_code(c only, run on linux)
- Filename : StudentID_lab9.c
- Input file : no

Lab9: B-Tree

- Note
  - Complete Search function, do not change other codes

Lab9: B-Tree

- Given ADT
- void Search(BTreeNode *Node, int Key);
  - Search Key
  - If Key is not found in Node and Node is Leaf, there are no Key in tree
  - If Key is not found in Node but Node is not Leaf, search recursively in the child

Lab9: B-Tree

```
Search 2
Key 2 exist in tree

Search 26
Key 26 exist in tree

Search 18
Key 18 exist in tree

Search 14
Key 14 not exist in tree
```