Practical 2
Implementation of B+ Tree

Gahan M. Saraiya, 18MCEC10

18mcec10@nirmauni.ac.in

I. Introduction

Aim of this practical is to implement algorithm of B+ tree

II. Implementation

I. Utility *utility.h*

```
//
// Created by jarvis on 17/8/18.
//

#ifndef DSA_LAB_UTILITY_H
#define DSA_LAB_UTILITY_H

#include <string.h>
#include <stdarg.h>

int max(int a, int b) { return (a > b)? a : b; }
int min(int a, int b) { return (a < b)? a : b; }

int write_log(const char *format, ...) {
    if(DEBUG) {
        printf("\n[DEBUG_LOG]> ");
        va_list args;
        va_start (args, format);
        vprintf(format, args);
        va_end (args);
    }
}

int *get_min_max(int *array, int no_of_elements, int min_max[]){
    // get minimum and maximum of array
//    printf("elements of array: ");
    for(int i=0; i<no_of_elements; i++){
//        printf("%d ", *(array + i));
        if (*(array + i) < min_max[0])
            min_max[0] = *(array + i);
```

```c
        if (*(array + i) > min_max[1])
            min_max[1] = *(array + i);
    }
    return min_max;
}

int display_array(int *array, int no_of_elements){
    // display given array of given size(no. of elements require because sizeof() returns max bound value)
    write_log(": ");
    for(int i=0; i<no_of_elements; i++){
        write_log( "%d ", *(array + i));
    }
    return 0;
}

int show_2d_array(int **array, int no_of_elements){
    // display given array of given size(no. of elements require because sizeof() returns max bound value)
    write_log(": ");
    for(int i=0; i<no_of_elements; i++){
        printf("a[%d][i]: ", i);
        for(int j=0; j<no_of_elements; j++) {
//            printf("array[%d][%d]: %d ", i, j, array[i][j]);
            printf("%d\t", array[i][j]);
        }
        printf("\twhere 0<=i<=%d\n", no_of_elements-1);
    }
    return 0;
}

int display_2d_array(int **array, int no_of_elements){
    // display given array of given size(no. of elements require because sizeof() returns max bound value)
    write_log(": ");
    for(int i=0; i<no_of_elements; i++){
        printf("a[%d][]: ", i);
        for(int j=0; j<no_of_elements; j++) {
//            printf("array[%d][%d]: %d ", i, j, array[i][j]);
            printf("%d ", array[i][j]);
        }
        printf("\n");
    }
    return 0;
}


void swap(int *one, int *two){
```

```c
    // swap function to swap elements by location/address
    int temp = *one;
    *one = *two;
    *two = temp;
}

#endif //DSA_LAB_UTILITY_H
```

II. Constants *constant.h*

```c
//
// Created by jarvis on 17/8/18.
//

#ifndef DSA_LAB_CONSTANT_H
#define DSA_LAB_CONSTANT_H
#define TEST_NUM 5000
#define DEBUG 0

#endif //DSA_LAB_CONSTANT_H
```

III. Main Program - *bPlusTree.c*

```c
//
// Created by Gahan Saraiya on 20/8/18.
// Implement B tree then B+ tree
// Implement B+ tree of degree 4
// Insertion, Search
//
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define DEBUG 0
#include "../utils/utility.h"

#define TREE_ORDER 4

typedef struct record {
    int value;
} record;

typedef struct Node {
    int total_keys;
    bool is_leaf;
    void **ptrs;
    int *keys;
```

```c
    struct Node *parent;
    struct Node *next;
} BPlusNode;

BPlusNode *insert_into_parent(BPlusNode *, BPlusNode *, int, BPlusNode *);
int exact_search(BPlusNode *root, int key);

record *NewRecord(int value) {
    record *new_record = (record *) malloc(sizeof(record));
    if (new_record == NULL) {
        perror("Record creation.");
        exit(EXIT_FAILURE);
    } else {
        new_record->value = value;
    }
    return new_record;
}

BPlusNode *find_leaf(BPlusNode *root, int key) {
//    Search Leaf Node for value -  key
    int i = 0;
    BPlusNode *n = root;
    if (n == NULL) {
        return n;
    }
    while (!n->is_leaf) {
        i = 0;
        while (i < n->total_keys) {
            if (key >= n->keys[i]) i++;
            else break;
        }
        n = (BPlusNode *) n->ptrs[i];
    }
    return n;
}

record *find(BPlusNode *root, int key) {
    int i = 0;
    BPlusNode *c = find_leaf(root, key);
    if (c == NULL) return NULL;
    for (i = 0; i < c->total_keys; i++)
        if (c->keys[i] == key) break;
    if (i == c->total_keys)
        return NULL;
    else
        return (record *) c->ptrs[i];
}

```

```
BPlusNode *newnode(void) {
    BPlusNode *new_node;
    new_node = (BPlusNode *) malloc(sizeof(BPlusNode));

    new_node->keys = (int *) malloc((TREE_ORDER - 1) * sizeof(int));

    new_node->ptrs = (void **) malloc(TREE_ORDER * sizeof(void *));

    new_node->is_leaf = false;
    new_node->total_keys = 0;
    new_node->parent = NULL;
    new_node->next = NULL;
    return new_node;
}

BPlusNode *insert_at_leaf(BPlusNode *leaf, int key, record *pointer) {

    int i, index;

    index = 0;
    while (index < leaf->total_keys && leaf->keys[index] < key)
        index++;

    for (i = leaf->total_keys; i > index; i--) {
        leaf->keys[i] = leaf->keys[i - 1];
        leaf->ptrs[i] = leaf->ptrs[i - 1];
    }
    leaf->keys[index] = key;
    leaf->ptrs[index] = pointer;
    leaf->total_keys++;
    return leaf;
}

BPlusNode *insert_into_node(BPlusNode *root, BPlusNode *n, int left_index, int
↪ key, BPlusNode *right) {
    int i;

    for (i = n->total_keys; i > left_index; i--) {
        n->ptrs[i + 1] = n->ptrs[i];
        n->keys[i] = n->keys[i - 1];
    }
    n->ptrs[left_index + 1] = right;
    n->keys[left_index] = key;
    n->total_keys++;
    return root;
}
```

```c
BPlusNode *insert_into_node_after_splitting(BPlusNode *root, BPlusNode *old_node, int left_index,
                                            int key, BPlusNode *right) {

    int i, j, s, k_prime;
    BPlusNode *new_node, *child;
    int *temp_keys;
    BPlusNode **temp_ptrs;

    temp_ptrs = (BPlusNode **) malloc((TREE_ORDER + 1) * sizeof(BPlusNode *));

    temp_keys = (int *) malloc(TREE_ORDER * sizeof(int));

    for (i = 0, j = 0; i < old_node->total_keys + 1; i++, j++) {
        if (j == left_index + 1) j++;
        temp_ptrs[j] = (BPlusNode *) old_node->ptrs[i];
    }

    for (i = 0, j = 0; i < old_node->total_keys; i++, j++) {
        if (j == left_index) j++;
        temp_keys[j] = old_node->keys[i];
    }

    temp_ptrs[left_index + 1] = right;
    temp_keys[left_index] = key;

    if (TREE_ORDER % 2 == 0)
        s = TREE_ORDER / 2;
    else
        s = TREE_ORDER / 2 + 1;

    new_node = newnode();

    old_node->total_keys = 0;
    for (i = 0; i < s - 1; i++) {
        old_node->ptrs[i] = temp_ptrs[i];
        old_node->keys[i] = temp_keys[i];
        old_node->total_keys++;
    }
    old_node->ptrs[i] = temp_ptrs[i];
    k_prime = temp_keys[s - 1];
    for (++i, j = 0; i < TREE_ORDER; i++, j++) {
        new_node->ptrs[j] = temp_ptrs[i];
        new_node->keys[j] = temp_keys[i];
        new_node->total_keys++;
    }
    new_node->ptrs[j] = temp_ptrs[i];

```

```
    new_node->parent = old_node->parent;
    for (i = 0; i <= new_node->total_keys; i++) {
        child = (BPlusNode *) new_node->ptrs[i];
        child->parent = new_node;
    }
    return insert_into_parent(root, old_node, k_prime, new_node);
}

BPlusNode *insert_into_parent(BPlusNode *root, BPlusNode *left, int key,
↪   BPlusNode *right) {

    int left_index;
    BPlusNode *parent;

    parent = left->parent;

    if (parent == NULL) {
        BPlusNode *r = newnode();
        r->keys[0] = key;
        r->ptrs[0] = left;
        r->ptrs[1] = right;
        r->total_keys++;
        r->parent = NULL;
        left->parent = r;
        right->parent = r;
        return r;
    }

    left_index = 0;

    while (left_index <= parent->total_keys && parent->ptrs[left_index] != left) {
    ↪   left_index++; }

    if (parent->total_keys < (TREE_ORDER - 1))
        return insert_into_node(root, parent, left_index, key, right);

    return insert_into_node_after_splitting(root, parent, left_index, key,
    ↪   right);
}

BPlusNode *split(BPlusNode *root, BPlusNode *leaf, int key, record *pointer) {
    BPlusNode *leaf_s;
    int *newkeys;
    void **newptrs;
    int insertindex, s, new_key, i, j;

    leaf_s = newnode();
    leaf_s->is_leaf = true;
```

```

    newkeys = (int *) malloc(TREE_ORDER * sizeof(int));

    newptrs = (void **) malloc(TREE_ORDER * sizeof(void *));

    insertindex = 0;
    while (insertindex < TREE_ORDER - 1 && leaf->keys[insertindex] < key)
        insertindex++;

    for (i = 0, j = 0; i < leaf->total_keys; i++, j++) {
        if (j == insertindex) j++;
        newkeys[j] = leaf->keys[i];
        newptrs[j] = leaf->ptrs[i];
    }

    newkeys[insertindex] = key;
    newptrs[insertindex] = pointer;

    leaf->total_keys = 0;

    if ((TREE_ORDER - 1) % 2 == 0)
        s = (TREE_ORDER - 1) / 2;
    else
        s = ((TREE_ORDER - 1) / 2) + 1;

    for (i = 0; i < s; i++) {
        leaf->ptrs[i] = newptrs[i];
        leaf->keys[i] = newkeys[i];
        leaf->total_keys++;
    }

    for (i = s, j = 0; i < TREE_ORDER; i++, j++) {
        leaf_s->ptrs[j] = newptrs[i];
        leaf_s->keys[j] = newkeys[i];
        leaf_s->total_keys++;
    }


    leaf_s->ptrs[TREE_ORDER - 1] = leaf->ptrs[TREE_ORDER - 1];//BPlusNode pointed by last pointer now should be pointed by new BPlusNode
    leaf->ptrs[TREE_ORDER - 1] = leaf_s;//new BPlusNode should be now pointed by previous BPlusNode

    for (i = leaf->total_keys; i < TREE_ORDER - 1; i++)//key holes in a BPlusNode
        leaf->ptrs[i] = NULL;
    for (i = leaf_s->total_keys; i < TREE_ORDER - 1; i++)
        leaf_s->ptrs[i] = NULL;//pointer holes in a BPlusNode

```

```
    leaf_s->parent = leaf->parent;
    new_key = leaf_s->keys[0];

    return insert_into_parent(root, leaf, new_key, leaf_s);
}

BPlusNode *insert(BPlusNode *root, int key, int value) {
    record *pointer;
    BPlusNode *leaf;

    if (root == NULL) {
//          Initializing Tree
        BPlusNode *l = newnode();

        l->is_leaf = true;
        root = l;
        root->keys[0] = key;
        root->ptrs[0] = pointer;
        root->ptrs[TREE_ORDER - 1] = NULL;
        root->parent = NULL;
        root->total_keys++;
        // write_log("\nRoot--> keys[0] = %d", root->keys[0]);
        return root;
    }

    if (find(root, key) != NULL)
        return root;

    pointer = NewRecord(value);
    leaf = find_leaf(root, key);

    if (leaf->total_keys < TREE_ORDER - 1) {
//          No splitting require as datum can be accommodate in free space
        leaf = insert_at_leaf(leaf, key, pointer);
        return root;
    }
    return split(root, leaf, key, pointer);
}

int path_to_root(BPlusNode *root, BPlusNode *child) {
    int length = 0;
    BPlusNode *c = child;
    while (c != root) {
        c = c->parent;
        length++;
    }
    return length;
}
```

```

BPlusNode *queue = NULL;

void Queue(BPlusNode *new_node) {
    BPlusNode *c;
    if (queue == NULL) {
        queue = new_node;
        queue->next = NULL;
    }
    else {
        c = queue;
        while (c->next != NULL) {
            c = c->next;
        }
        c->next = new_node;
        new_node->next = NULL;
    }
}

BPlusNode *deQueue(void) {
    BPlusNode *n = queue;
    queue = queue->next;
    n->next = NULL;
    return n;
}

void pretty_print(BPlusNode *root) {
    write_log("Printing Tree\n");
    BPlusNode *n = NULL;
    int i = 0;
    int rank = 0;
    int new_rank = 0;

    if (root == NULL) {
        printf("\nOpsss... It seems no value exist, Kindly consider adding
         ↪ element(s)\n");
        return;
    }

    queue = NULL;
    Queue(root);
    while (queue != NULL) {
        n = deQueue();
        if (n->parent != NULL && n == n->parent->ptrs[0]) {
            new_rank = path_to_root(root, n);
            if (new_rank != rank) {
                rank = new_rank;
                printf("\n");
```

```
            }
        }

        for (i = 0; i < n->total_keys; i++) {
            printf("%d ", n->keys[i]);
        }
        if (!n->is_leaf) {
            for (i = 0; i <= n->total_keys; i++)
                Queue((BPlusNode *) n->ptrs[i]);
        }
        printf(" | ");
    }
    printf("\n");
}

int cut(int length) {
    if (length % 2 == 0)
        return length / 2;
    else
        return length / 2 + 1;
}


int get_neighbor_index(BPlusNode *n) {

    int i;
    for (i = 0; i <= n->parent->total_keys; i++)
        if (n->parent->ptrs[i] == n)
            // return neighbouring node.
            return i - 1;

    printf("Search for non-existent pointer to BPlusNode in parent.\n");
    printf("Node:  %#lx\n", (unsigned long) n);
    exit(EXIT_FAILURE);
}

int search(BPlusNode *root, int key) {
    write_log("In batch search");
    int i = 0, match = 0;
    //------------------------first find in leaf BPlusNode is the key is
     ↪ found.---------
    BPlusNode *c = find_leaf(root, key);
    if (c == NULL) {
        // data/key not exist
        match = 0;
    }
    for (i = 0; i < c->total_keys; i++) {
        if (c->keys[i] == key) {
```

```
            // data found
            match = 1;
            break;
        }
    }
    return match;
}

int batch_search(BPlusNode *root) {
    write_log("In batch search");

    int start, end, flag = 1;
    BPlusNode *n = NULL;
    int i = 0, rank = 0, new_rank = 0;
    int exact_match_flag = 0;
    printf("\nstart value: ");
    scanf("%d", &start);
    printf("\nend value: ");
    scanf("%d", &end);

    queue = NULL;
    Queue(root);
    while (queue != NULL) {
        n = deQueue();
        if (n->parent != NULL && n == n->parent->ptrs[0]) {
            new_rank = path_to_root(root, n);
            if (new_rank != rank) {
                rank = new_rank;
                printf("Depth level: %d", rank);
                printf("\n");
            }
        }

        for (i = 0; i < n->total_keys; i++) {
            if (n->is_leaf && n->keys[i] >= start && n->keys[i] <= end) {
                if (flag) {
                    write_log("Traversed neighbour\n");
                    flag = 0;
                }
                printf("%d ", n->keys[i]);
            }
        }
        if (!n->is_leaf) {
            for (i = 0; i <= n->total_keys; i++)
                Queue((BPlusNode *) n->ptrs[i]);
        }
    }
    return 0;
```

```
}


int main(int argc, char *argv[]) {
//    int degree;
//    if (atoi(argv[1]))
//        degree = atoi(argv[1]);
//    else
//        degree = TREE_ORDER;
    int find_key, batch_search_value[100], n, i = 0, max, min;
    BPlusNode *root;
    root = NULL;

    printf("\nB+ Tree Degree (must be at least 3): %d", TREE_ORDER);
    printf("\n########################################################"
           "\n1. Insert"
           "\n2. Search"
           "\n3. Batch Search"
           "\n4. Print Tree"
           "\n5. Exit"
           "\n########################################################\n");
    int choice;
    while (choice != 5){
        printf("choice: ");
        scanf("%d", &choice);
        int value, result;
        int start, end;
        switch (choice) {
            case 1:
                printf("\nValue: ");
                scanf("%d", &value);
                root = insert(root, value, value);
                printf("\nB+ Tree : \n");
                pretty_print(root);
                break;
            case 2:
                printf("\nSearch Value: ");
                scanf("%d", &value);
                result = search(root, value);
                if (result)
                    printf("Value %d matched\n", value);
                else
                    printf("Value %d does not exist\n", value);
                break;
            case 3:
                printf("\nBatch Search: ");
                result = batch_search(root);
                printf("\n");
```

```
                break;
            case 4:
                pretty_print(root);
                break;
            case 5:
                printf("\nGreetings!!! see you later...\n");
                return 0;
            default:
                printf("\nKindly select correct value...\n");
                printf("\n######################################################"
                       "\n1. Insert"
                       "\n2. Search"
                       "\n3. Batch Search"
                       "\n4. Print Tree"
                       "\n5. Exit"
                       
                       ↪ "\n######################################################\n");
        }
    }

    return 0;
}
```

Output

```
B+ Tree Degree (must be at least 3): 4
######################################################
1. Insert
2. Search
3. Batch Search
4. Print Tree
5. Exit
######################################################
choice: 1

Value: 5

B+ Tree :
5  |
choice: 1

Value: 59

B+ Tree :
```

```
5 59  |
choice: 1

Value: 66

B+ Tree :
5 59 66  |
choice: 1

Value: 14

B+ Tree :
59  |
5 14  | 59 66  |
choice: 1

Value: 98

B+ Tree :
59  |
5 14  | 59 66 98  |
choice: 1

Value: 105

B+ Tree :
59 98  |
5 14  | 59 66  | 98 105  |
choice: 1

Value: 1500

B+ Tree :
59 98  |
5 14  | 59 66  | 98 105 1500  |
choice: 1

Value: 1109

B+ Tree :
59 98 1109  |
```

```
5 14  | 59 66  | 98 105  | 1109 1500  |
choice: 1

Value: 23

B+ Tree :
59 98 1109  |
5 14 23  | 59 66  | 98 105  | 1109 1500  |
choice: 1

Value: 50

B+ Tree :
59  |
23  | 98 1109  |
5 14  | 23 50  | 59 66  | 98 105  | 1109 1500  |
choice: 1

Value: 109

B+ Tree :
59  |
23  | 98 1109  |
5 14  | 23 50  | 59 66  | 98 105 109  | 1109 1500  |
choice: 1

Value: 90

B+ Tree :
59  |
23  | 98 1109  |
5 14  | 23 50  | 59 66 90  | 98 105 109  | 1109 1500  |
choice: 1

Value: 51

B+ Tree :
59  |
23  | 98 1109  |
5 14  | 23 50 51  | 59 66 90  | 98 105 109  | 1109 1500  |
choice: 1
```

```
Value: 52

B+ Tree :
59  |
23 51  | 98 1109  |
5 14  | 23 50  | 51 52  | 59 66 90  | 98 105 109  | 1109 1500  |
choice: 1

Value: 25

B+ Tree :
59  |
23 51  | 98 1109  |
5 14  | 23 25 50  | 51 52  | 59 66 90  | 98 105 109  | 1109 1500  |
choice: 1

Value: 26

B+ Tree :
59  |
23 26 51  | 98 1109  |
5 14  | 23 25  | 26 50  | 51 52  | 59 66 90  | 98 105 109  | 1109 1500  |
choice: 1

Value: 27

B+ Tree :
59  |
23 26 51  | 98 1109  |
5 14  | 23 25  | 26 27 50  | 51 52  | 59 66 90  | 98 105 109  | 1109 1500  |
choice: 1

Value: 28

B+ Tree :
26 59  |
23  | 28 51  | 98 1109  |
5 14  | 23 25  | 26 27  | 28 50  | 51 52  | 59 66 90  | 98 105 109  | 1109 1500
 ↪  |
choice: 1
```

```
Value: 100

B+ Tree :
26 59  |
23  | 28 51  | 98 105 1109  |
5 14  | 23 25  | 26 27  | 28 50  | 51 52  | 59 66 90  | 98 100  | 105 109  |
 ↪  1109 1500  |
choice: 1

Value: 92

B+ Tree :
26 59 98  |
23  | 28 51  | 90  | 105 1109  |
5 14  | 23 25  | 26 27  | 28 50  | 51 52  | 59 66  | 90 92  | 98 100  | 105 109
 ↪  | 1109 1500  |
choice: 1

Value: 53

B+ Tree :
26 59 98  |
23  | 28 51  | 90  | 105 1109  |
5 14  | 23 25  | 26 27  | 28 50  | 51 52 53  | 59 66  | 90 92  | 98 100  | 105
 ↪  109  | 1109 1500  |
choice: 1

Value: 17

B+ Tree :
26 59 98  |
23  | 28 51  | 90  | 105 1109  |
5 14 17  | 23 25  | 26 27  | 28 50  | 51 52 53  | 59 66  | 90 92  | 98 100  |
 ↪  105 109  | 1109 1500  |
choice: 1

Value: 1

B+ Tree :
26 59 98  |
```

```
14 23  | 28 51  | 90  | 105 1109  |
1 5  | 14 17  | 23 25  | 26 27  | 28 50  | 51 52 53  | 59 66  | 90 92  | 98 100
↪  | 105 109  | 1109 1500  |
choice: 1

Value: 0

B+ Tree :
26 59 98  |
14 23  | 28 51  | 90  | 105 1109  |
0 1 5  | 14 17  | 23 25  | 26 27  | 28 50  | 51 52 53  | 59 66  | 90 92  | 98
↪  100  | 105 109  | 1109 1500  |
choice: 1

Value: 103

B+ Tree :
26 59 98  |
14 23  | 28 51  | 90  | 105 1109  |
0 1 5  | 14 17  | 23 25  | 26 27  | 28 50  | 51 52 53  | 59 66  | 90 92  | 98
↪  100 103  | 105 109  | 1109 1500  |
choice: 1

Value: 108

B+ Tree :
26 59 98  |
14 23  | 28 51  | 90  | 105 1109  |
0 1 5  | 14 17  | 23 25  | 26 27  | 28 50  | 51 52 53  | 59 66  | 90 92  | 98
↪  100 103  | 105 108 109  | 1109 1500  |
choice: 2

Search Value: 52
Value 52 matched
choice: 2

Search Value: 555
Value 555 does not exist
choice: 3

Batch Search:
```

```
start value: 5

end value: 52
Depth level: 1
Depth level: 2
5 14 17 23 25 26 27 28 50 51 52
choice: 4
26 59 98  |
14 23  | 28 51  | 90  | 105 1109  |
0 1 5  | 14 17  | 23 25  | 26 27  | 28 50  | 51 52 53  | 59 66  | 90 92  | 98
 ↪  100 103  | 105 108 109  | 1109 1500  |
choice: 5

Greetings!!! see you later...
```

III. Summary

- all leaves at the same lowest level
- all nodes at least half full (except root)

Let f be the degree of tree and n be the total number of data then

Space Complexity

	Max # pointers	**Max # keys**	**Min # pointers**	**Min # keys**
Non-leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f-1$	2	1
Leaf	f	$f-1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

- Number of disk accesses proportional to the height of the B-tree. - The ***worst-case height*** of a B+ tree is:

Let f be the degree of tree and n be the total number of data then

Space Complexity

	Time Complexity	**Remarks**
height	$O(\log_f n)$	
Root	$O(f \log_f n)$	linear search inside each nodes
search	$O(\log_2 f \log_f n)$	binary search inside each node
insert	$O(\log_f n)$	if splitting not require
insert	$O(f \log_f n)$	if splitting require
insert	$O(\log_f n)$	if merge not require
insert	$O(f \log_f n)$	if merge require