

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*

```
1 //
2 // Created by jarvis on 17/8/18.
3 //
4
5 #ifndef DSA_LAB_UTILITY_H
6 #define DSA_LAB_UTILITY_H
7
8 #include <string.h>
9 #include <stdarg.h>
10
11 int max(int a, int b) { return (a > b)? a : b; }
12 int min(int a, int b) { return (a < b)? a : b; }
13
14 int write_log(const char *format, ...) {
15     if(DEBUG) {
16         printf("\n[DEBUG_LOG]> ");
17         va_list args;
18         va_start (args, format);
19         vprintf(format, args);
20         va_end (args);
21     }
22 }
23
24 int *get_min_max(int *array, int no_of_elements, int min_max[]){
25     // get minimum and maximum of array
26     printf("elements of array: ");
27     for(int i=0; i<no_of_elements; i++){
28         printf("%d ", *(array + i));
29         if (*(array + i) < min_max[0])
30             min_max[0] = *(array + i);
```

```

31         if (*(array + i) > min_max[1])
32             min_max[1] = *(array + i);
33     }
34     return min_max;
35 }
36
37 int display_array(int *array, int no_of_elements){
38     // display given array of given size(no. of elements require because sizeof()
39     // returns max bound value)
40     write_log(": ");
41     for(int i=0; i<no_of_elements; i++){
42         write_log(" %d ", *(array + i));
43     }
44     return 0;
45 }
46
47 int show_2d_array(int **array, int no_of_elements){
48     // display given array of given size(no. of elements require because sizeof()
49     // returns max bound value)
50     write_log(": ");
51     for(int i=0; i<no_of_elements; i++){
52         printf("a[%d][i]: ", i);
53         for(int j=0; j<no_of_elements; j++) {
54             // printf("array[%d][%d]: %d ", i, j, array[i][j]);
55             printf("%d\t", array[i][j]);
56         }
57         printf("\twhere 0<=i<=%d\n", no_of_elements-1);
58     }
59     return 0;
60 }
61
62 int display_2d_array(int **array, int no_of_elements){
63     // display given array of given size(no. of elements require because sizeof()
64     // returns max bound value)
65     write_log(": ");
66     for(int i=0; i<no_of_elements; i++){
67         printf("a[%d][i]: ", i);
68         for(int j=0; j<no_of_elements; j++) {
69             // printf("array[%d][%d]: %d ", i, j, array[i][j]);
70             printf("%d ", array[i][j]);
71         }
72         printf("\n");
73     }
74     return 0;
75 }
76
77 void swap(int *one, int *two){

```

```

76     // swap function to swap elements by location/address
77     int temp = *one;
78     *one = *two;
79     *two = temp;
80 }
81
82 #endif //DSA_LAB_UTILITY_H

```

II. Constants *constant.h*

```

1 //
2 // Created by jarvis on 17/8/18.
3 //
4
5 #ifndef DSA_LAB_CONSTANT_H
6 #define DSA_LAB_CONSTANT_H
7 #define TEST_NUM 5000
8 #define DEBUG 0
9
10 #endif //DSA_LAB_CONSTANT_H

```

III. Main Program - *bPlusTree.c*

```

1 //
2 // Created by Gahan Saraiya on 20/8/18.
3 // Implement B tree then B+ tree
4 // Implement B+ tree of degree 4
5 // Insertion, Search
6 //
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <stdbool.h>
10 #define DEBUG 0
11 #include "../utils/utility.h"
12
13 #define TREE_ORDER 4
14
15 typedef struct record {
16     int value;
17 } record;
18
19 typedef struct Node {
20     int total_keys;
21     bool is_leaf;
22     void **ptrs;
23     int *keys;

```

```

24     struct Node *parent;
25     struct Node *next;
26 } BPlusNode;
27
28 BPlusNode *insert_into_parent(BPlusNode *, BPlusNode *, int, BPlusNode *);
29 int exact_search(BPlusNode *root, int key);
30
31 record *NewRecord(int value) {
32     record *new_record = (record *) malloc(sizeof(record));
33     if (new_record == NULL) {
34         perror("Record creation.");
35         exit(EXIT_FAILURE);
36     } else {
37         new_record->value = value;
38     }
39     return new_record;
40 }
41
42 BPlusNode *find_leaf(BPlusNode *root, int key) {
43     // Search Leaf Node for value - key
44     int i = 0;
45     BPlusNode *n = root;
46     if (n == NULL) {
47         return n;
48     }
49     while (!n->is_leaf) {
50         i = 0;
51         while (i < n->total_keys) {
52             if (key >= n->keys[i]) i++;
53             else break;
54         }
55         n = (BPlusNode *) n->ptrs[i];
56     }
57     return n;
58 }
59
60 record *find(BPlusNode *root, int key) {
61     int i = 0;
62     BPlusNode *c = find_leaf(root, key);
63     if (c == NULL) return NULL;
64     for (i = 0; i < c->total_keys; i++)
65         if (c->keys[i] == key) break;
66     if (i == c->total_keys)
67         return NULL;
68     else
69         return (record *) c->ptrs[i];
70 }
71

```

```

72
73 BPlusNode *newnode(void) {
74     BPlusNode *new_node;
75     new_node = (BPlusNode *) malloc(sizeof(BPlusNode));
76
77     new_node->keys = (int *) malloc((TREE_ORDER - 1) * sizeof(int));
78
79     new_node->ptrs = (void **) malloc(TREE_ORDER * sizeof(void *));
80
81     new_node->is_leaf = false;
82     new_node->total_keys = 0;
83     new_node->parent = NULL;
84     new_node->next = NULL;
85     return new_node;
86 }
87
88 BPlusNode *insert_at_leaf(BPlusNode *leaf, int key, record *pointer) {
89
90     int i, index;
91
92     index = 0;
93     while (index < leaf->total_keys && leaf->keys[index] < key)
94         index++;
95
96     for (i = leaf->total_keys; i > index; i--) {
97         leaf->keys[i] = leaf->keys[i - 1];
98         leaf->ptrs[i] = leaf->ptrs[i - 1];
99     }
100     leaf->keys[index] = key;
101     leaf->ptrs[index] = pointer;
102     leaf->total_keys++;
103     return leaf;
104 }
105
106 BPlusNode *insert_into_node(BPlusNode *root, BPlusNode *n, int left_index, int
    ⇨ key, BPlusNode *right) {
107     int i;
108
109     for (i = n->total_keys; i > left_index; i--) {
110         n->ptrs[i + 1] = n->ptrs[i];
111         n->keys[i] = n->keys[i - 1];
112     }
113     n->ptrs[left_index + 1] = right;
114     n->keys[left_index] = key;
115     n->total_keys++;
116     return root;
117 }
118

```

```
119 BPlusNode *insert_into_node_after_splitting(BPlusNode *root, BPlusNode *old_node,
    ↳ int left_index,
120                                     int key, BPlusNode *right) {
121
122     int i, j, s, k_prime;
123     BPlusNode *new_node, *child;
124     int *temp_keys;
125     BPlusNode **temp_ptrs;
126
127     temp_ptrs = (BPlusNode **) malloc((TREE_ORDER + 1) * sizeof(BPlusNode *));
128
129     temp_keys = (int *) malloc(TREE_ORDER * sizeof(int));
130
131     for (i = 0, j = 0; i < old_node->total_keys + 1; i++, j++) {
132         if (j == left_index + 1) j++;
133         temp_ptrs[j] = (BPlusNode *) old_node->ptrs[i];
134     }
135
136     for (i = 0, j = 0; i < old_node->total_keys; i++, j++) {
137         if (j == left_index) j++;
138         temp_keys[j] = old_node->keys[i];
139     }
140
141     temp_ptrs[left_index + 1] = right;
142     temp_keys[left_index] = key;
143
144     if (TREE_ORDER % 2 == 0)
145         s = TREE_ORDER / 2;
146     else
147         s = TREE_ORDER / 2 + 1;
148
149     new_node = newnode();
150
151     old_node->total_keys = 0;
152     for (i = 0; i < s - 1; i++) {
153         old_node->ptrs[i] = temp_ptrs[i];
154         old_node->keys[i] = temp_keys[i];
155         old_node->total_keys++;
156     }
157     old_node->ptrs[i] = temp_ptrs[i];
158     k_prime = temp_keys[s - 1];
159     for (++i, j = 0; i < TREE_ORDER; i++, j++) {
160         new_node->ptrs[j] = temp_ptrs[i];
161         new_node->keys[j] = temp_keys[i];
162         new_node->total_keys++;
163     }
164     new_node->ptrs[j] = temp_ptrs[i];
165
```

```

166     new_node->parent = old_node->parent;
167     for (i = 0; i <= new_node->total_keys; i++) {
168         child = (BPlusNode *) new_node->ptrs[i];
169         child->parent = new_node;
170     }
171     return insert_into_parent(root, old_node, k_prime, new_node);
172 }
173
174 BPlusNode *insert_into_parent(BPlusNode *root, BPlusNode *left, int key,
    ↪ BPlusNode *right) {
175
176     int left_index;
177     BPlusNode *parent;
178
179     parent = left->parent;
180
181     if (parent == NULL) {
182         BPlusNode *r = newnode();
183         r->keys[0] = key;
184         r->ptrs[0] = left;
185         r->ptrs[1] = right;
186         r->total_keys++;
187         r->parent = NULL;
188         left->parent = r;
189         right->parent = r;
190         return r;
191     }
192
193     left_index = 0;
194
195     while (left_index <= parent->total_keys && parent->ptrs[left_index] != left) {
    ↪ left_index++; }
196
197     if (parent->total_keys < (TREE_ORDER - 1))
198         return insert_into_node(root, parent, left_index, key, right);
199
200     return insert_into_node_after_splitting(root, parent, left_index, key,
    ↪ right);
201 }
202
203 BPlusNode *split(BPlusNode *root, BPlusNode *leaf, int key, record *pointer) {
204     BPlusNode *leaf_s;
205     int *newkeys;
206     void **newptrs;
207     int insertindex, s, new_key, i, j;
208
209     leaf_s = newnode();
210     leaf_s->is_leaf = true;

```

```

211
212     newkeys = (int *) malloc(TREE_ORDER * sizeof(int));
213
214     newptrs = (void **) malloc(TREE_ORDER * sizeof(void *));
215
216     insertindex = 0;
217     while (insertindex < TREE_ORDER - 1 && leaf->keys[insertindex] < key)
218         insertindex++;
219
220     for (i = 0, j = 0; i < leaf->total_keys; i++, j++) {
221         if (j == insertindex) j++;
222         newkeys[j] = leaf->keys[i];
223         newptrs[j] = leaf->ptrs[i];
224     }
225
226     newkeys[insertindex] = key;
227     newptrs[insertindex] = pointer;
228
229     leaf->total_keys = 0;
230
231     if ((TREE_ORDER - 1) % 2 == 0)
232         s = (TREE_ORDER - 1) / 2;
233     else
234         s = ((TREE_ORDER - 1) / 2) + 1;
235
236     for (i = 0; i < s; i++) {
237         leaf->ptrs[i] = newptrs[i];
238         leaf->keys[i] = newkeys[i];
239         leaf->total_keys++;
240     }
241
242     for (i = s, j = 0; i < TREE_ORDER; i++, j++) {
243         leaf_s->ptrs[j] = newptrs[i];
244         leaf_s->keys[j] = newkeys[i];
245         leaf_s->total_keys++;
246     }
247
248
249     leaf_s->ptrs[TREE_ORDER - 1] = leaf->ptrs[TREE_ORDER - 1]; //BPlusNode pointed
    ↪ by last pointer now should be pointed by new BPlusNode
250     leaf->ptrs[TREE_ORDER - 1] = leaf_s; //new BPlusNode should be now pointed by
    ↪ previous BPlusNode
251
252     for (i = leaf->total_keys; i < TREE_ORDER - 1; i++) //key holes in a BPlusNode
253         leaf->ptrs[i] = NULL;
254     for (i = leaf_s->total_keys; i < TREE_ORDER - 1; i++)
255         leaf_s->ptrs[i] = NULL; //pointer holes in a BPlusNode
256

```



```
257     leaf_s->parent = leaf->parent;
258     new_key = leaf_s->keys[0];
259
260     return insert_into_parent(root, leaf, new_key, leaf_s);
261 }
262
263 BPlusNode *insert(BPlusNode *root, int key, int value) {
264     record *pointer;
265     BPlusNode *leaf;
266
267     if (root == NULL) {
268         //      Initializing Tree
269         BPlusNode *l = newnode();
270
271         l->is_leaf = true;
272         root = l;
273         root->keys[0] = key;
274         root->ptrs[0] = pointer;
275         root->ptrs[TREE_ORDER - 1] = NULL;
276         root->parent = NULL;
277         root->total_keys++;
278         // write_log("\nRoot--> keys[0] = %d", root->keys[0]);
279         return root;
280     }
281
282     if (find(root, key) != NULL)
283         return root;
284
285     pointer = NewRecord(value);
286     leaf = find_leaf(root, key);
287
288     if (leaf->total_keys < TREE_ORDER - 1) {
289         //      No splitting require as datum can be accommodate in free space
290         leaf = insert_at_leaf(leaf, key, pointer);
291         return root;
292     }
293     return split(root, leaf, key, pointer);
294 }
295
296 int path_to_root(BPlusNode *root, BPlusNode *child) {
297     int length = 0;
298     BPlusNode *c = child;
299     while (c != root) {
300         c = c->parent;
301         length++;
302     }
303     return length;
304 }
```

```
305
306 BPlusNode *queue = NULL;
307
308 void Queue(BPlusNode *new_node) {
309     BPlusNode *c;
310     if (queue == NULL) {
311         queue = new_node;
312         queue->next = NULL;
313     }
314     else {
315         c = queue;
316         while (c->next != NULL) {
317             c = c->next;
318         }
319         c->next = new_node;
320         new_node->next = NULL;
321     }
322 }
323
324 BPlusNode *deQueue(void) {
325     BPlusNode *n = queue;
326     queue = queue->next;
327     n->next = NULL;
328     return n;
329 }
330
331 void pretty_print(BPlusNode *root) {
332     write_log("Printing Tree\n");
333     BPlusNode *n = NULL;
334     int i = 0;
335     int rank = 0;
336     int new_rank = 0;
337
338     if (root == NULL) {
339         printf("\nOpsss... It seems no value exist, Kindly consider adding
340             ↪ element(s)\n");
341         return;
342     }
343
344     queue = NULL;
345     Queue(root);
346     while (queue != NULL) {
347         n = deQueue();
348         if (n->parent != NULL && n == n->parent->ptrs[0]) {
349             new_rank = path_to_root(root, n);
350             if (new_rank != rank) {
351                 rank = new_rank;
352                 printf("\n");
353             }
354         }
355     }
356 }
```

```

352     }
353 }
354
355     for (i = 0; i < n->total_keys; i++) {
356         printf("%d ", n->keys[i]);
357     }
358     if (!n->is_leaf) {
359         for (i = 0; i <= n->total_keys; i++)
360             Queue((BPlusNode *) n->ptrs[i]);
361     }
362     printf(" | ");
363 }
364 printf("\n");
365 }
366
367 int cut(int length) {
368     if (length % 2 == 0)
369         return length / 2;
370     else
371         return length / 2 + 1;
372 }
373
374
375 int get_neighbor_index(BPlusNode *n) {
376
377     int i;
378     for (i = 0; i <= n->parent->total_keys; i++)
379         if (n->parent->ptrs[i] == n)
380             // return neighbouring node.
381             return i - 1;
382
383     printf("Search for non-existent pointer to BPlusNode in parent.\n");
384     printf("Node:  %#lx\n", (unsigned long) n);
385     exit(EXIT_FAILURE);
386 }
387
388 int search(BPlusNode *root, int key) {
389     write_log("In batch search");
390     int i = 0, match = 0;
391     //-----first find in leaf BPlusNode is the key is
392     //    found.-----
393     BPlusNode *c = find_leaf(root, key);
394     if (c == NULL) {
395         // data/key not exist
396         match = 0;
397     }
398     for (i = 0; i < c->total_keys; i++) {
399         if (c->keys[i] == key) {

```

```

399         // data found
400         match = 1;
401         break;
402     }
403 }
404 return match;
405 }
406
407 int batch_search(BPlusNode *root) {
408     write_log("In batch search");
409
410     int start, end, flag = 1;
411     BPlusNode *n = NULL;
412     int i = 0, rank = 0, new_rank = 0;
413     int exact_match_flag = 0;
414     printf("\nstart value: ");
415     scanf("%d", &start);
416     printf("\nend value: ");
417     scanf("%d", &end);
418
419     queue = NULL;
420     Queue(root);
421     while (queue != NULL) {
422         n = deQueue();
423         if (n->parent != NULL && n == n->parent->ptrs[0]) {
424             new_rank = path_to_root(root, n);
425             if (new_rank != rank) {
426                 rank = new_rank;
427                 printf("Depth level: %d", rank);
428                 printf("\n");
429             }
430         }
431
432         for (i = 0; i < n->total_keys; i++) {
433             if (n->is_leaf && n->keys[i] >= start && n->keys[i] <= end) {
434                 if (flag) {
435                     write_log("Traversed neighbour\n");
436                     flag = 0;
437                 }
438                 printf("%d ", n->keys[i]);
439             }
440         }
441         if (!n->is_leaf) {
442             for (i = 0; i <= n->total_keys; i++)
443                 Queue((BPlusNode *) n->ptrs[i]);
444         }
445     }
446     return 0;

```

```
447 }
448
449
450 int main(int argc, char *argv[]) {
451     // int degree;
452     // if (atoi(argv[1]))
453     //     degree = atoi(argv[1]);
454     // else
455     //     degree = TREE_ORDER;
456     int find_key, batch_search_value[100], n, i = 0, max, min;
457     BPlusNode *root;
458     root = NULL;
459
460     printf("\nB+ Tree Degree (must be at least 3): %d", TREE_ORDER);
461     printf("\n#####\n1. Insert\n2. Search\n3. Batch Search\n4. Print Tree\n5. Exit\n#####\n");
462     int choice;
463     while (choice != 5){
464         printf("choice: ");
465         scanf("%d", &choice);
466         int value, result;
467         int start, end;
468         switch (choice) {
469             case 1:
470                 printf("\nValue: ");
471                 scanf("%d", &value);
472                 root = insert(root, value, value);
473                 printf("\nB+ Tree : \n");
474                 pretty_print(root);
475                 break;
476             case 2:
477                 printf("\nSearch Value: ");
478                 scanf("%d", &value);
479                 result = search(root, value);
480                 if (result)
481                     printf("Value %d matched\n", value);
482                 else
483                     printf("Value %d does not exist\n", value);
484                 break;
485             case 3:
486                 printf("\nBatch Search: ");
487                 result = batch_search(root);
488                 printf("\n");
489         }
490     }
```

```

495         break;
496     case 4:
497         pretty_print(root);
498         break;
499     case 5:
500         printf("\nGreetings!!! see you later...\n");
501         return 0;
502     default:
503         printf("\nKindly select correct value...\n");
504         printf("\n#####")
505             "\n1. Insert"
506             "\n2. Search"
507             "\n3. Batch Search"
508             "\n4. Print Tree"
509             "\n5. Exit"
510             "\n#####\n");
511     }
512 }
513
514 return 0;
515 }

```

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 |