

# Data Structures

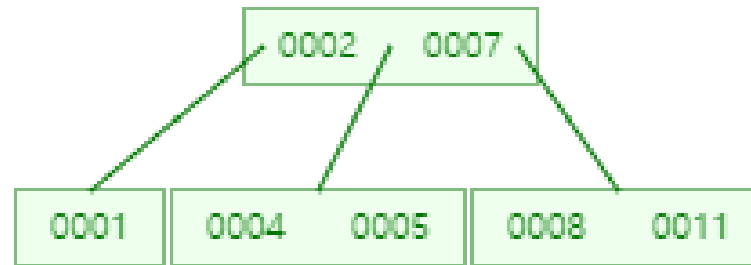
박영준 교수님

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\_DEGREE \* 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;  
};

# 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 child has half of `NODE_DEGREE` keys
  - Link separated child 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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define TRUE 1
5 #define FALSE 0
6
7 #define NODE_DEGREE      2 //Order
8 #define MAX_CHILDS      (NODE_DEGREE * 2)
9 #define MAX_KEYS         (MAX_CHILDS - 1)
10
11 #define SPACE (4 * MAX_KEYS)
12
13 typedef struct BTreeNode
14 {
15     int Keys[MAX_KEYS];
16     struct BTreeNode *Childs[MAX_CHILDS];
17     int KeyIndex;
18     int Leaf;
19 } BTreeNode;
20
21 typedef struct
22 {
23     struct BTreeNode *Root;
24     int Degree;
25 } BTree;
26
```

```
28 BTreeNode *CreateBTreeNode();
29 BTree *CreateBTree();
30
31 void Search( BTreeNode *Node, int Key);
32
33 void SplitChild(BTreeNode *Parent, BTreeNode *Child, int Index);
34
35 void InsertNonFull(BTreeNode *Node, int Key);
36 void Insert(BTree *Tree, int Key);
37
38 void Remove(BTree *Tree, int Key);
39
40 int FindKey(BTreeNode *Node, int Key);
41 void RemoveNode(BTreeNode *Node, int Key);
42
43 void RemoveFromLeaf(BTreeNode *Node, int Index);
44 void RemoveFromNonLeaf(BTreeNode *Node, int Index);
45
46 void Fill(BTreeNode *Node, int Index);
47
48 int RetPredecessor(BTreeNode *Node, int Index);
49 int RetSuccessor(BTreeNode *Node, int Index);
50
51 void BorrowFromPrev(BTreeNode *Node, int Index);
52 void BorrowFromNext(BTreeNode *Node, int Index);
53
54 void Merge(BTreeNode *Node, int Index);
55
56 void PrintTree(BTreeNode *Node, int Level, int Blank);
--
```



# B-Tree

```
262 void SplitChild(BTreeNode *Parent, BTreeNode *Child, int Index)
263 {
264     //temporary stores Child
265     BTreeNode *temp = CreateBTreeNode();
266
267     temp->Leaf = Child->Leaf;
268     temp->KeyIndex = NODE_DEGREE - 1;
269
270     //move right half keys of Child to the temp
271     for(int i = 0; i < NODE_DEGREE - 1; i++)
272     {
273         temp->Keys[i] = Child->Keys[NODE_DEGREE + i];
274     }
275
276     //if child is not leaf, move childs of child to the temp
277     if(Child->Leaf == FALSE)
278     {
279         for(int i = 0; i < NODE_DEGREE; i++)
280         {
281             temp->Childs[i] = Child->Childs[NODE_DEGREE + i];
282         }
283     }
284     Child->KeyIndex = NODE_DEGREE - 1;
285
286     //move childs of parent, make space to insert temp
287     for(int i = Parent->KeyIndex; i > Index - 1; i--)
288     {
289         Parent->Childs[i + 1] = Parent->Childs[i];
290     }
291
292     //insert temp
293     Parent->Childs[Index] = temp;
294
295     //move keys of parent, make space to insert keys of child
296     for(int i = Parent->KeyIndex; i > Index - 1; i--)
297     {
298         Parent->Keys[i] = Parent->Keys[i - 1];
299     }
300     Parent->Keys[Index - 1] = Child->Keys[NODE_DEGREE - 1];
301
302     Parent->KeyIndex++;
303 }
```

# B-Tree

```
305 void InsertNonFull(BTreeNode *Node, int Key)
306 {
307     int Index = Node->KeyIndex;
308
309     if(Node->Leaf == TRUE)
310     {
311         //find location to insert key
312         while(Index > 0 && Node->Keys[Index - 1] > Key)
313         {
314             Node->Keys[Index] = Node->Keys[Index - 1];
315             Index--;
316         }
317         Node->Keys[Index] = Key;
318         Node->KeyIndex++;
319     }
320     else
321     {
322         //find location to insert key
323         while(Index > 0 && Node->Keys[Index - 1] > Key)
324         {
325             Index--;
326         }
327
328         if(Node->Childs[Index]->KeyIndex == MAX_KEYS)
329         {
330             //if childs is full, split
331             SplitChild(Node, Node->Childs[Index], Index + 1);
332
333             //find child index to insert key
334             if(Node->Keys[Index] < Key)
335             {
336                 Index++;
337             }
338         }
339         InsertNonFull(Node->Childs[Index], Key);
340     }
341 }
```

# B-Tree

```
343 void Insert(BTree *Tree, int Key)
344 {
345     if(Tree->Root->KeyIndex == MAX_KEYS)
346     {
347         //if root is full, split
348         BTreeNode *temp = CreateBTreeNode();
349
350         temp->Leaf = FALSE;
351         temp->KeyIndex = 0;
352         temp->Childs[0] = Tree->Root;
353
354         //make root is first child of new root
355         SplitChild(temp, Tree->Root, 1);
356
357         //insert key
358         InsertNonFull(temp, Key);
359
360         //change root pointer to new root
361         Tree->Root = temp;
362     }
363     else
364     {
365         //if root is not full, insert key
366         InsertNonFull(Tree->Root, Key);
367     }
368 }
```

```
370 void Remove(BTree *Tree, int Key)
371 {
372     //remove key first
373     RemoveNode(Tree->Root, Key);
374
375     if(Tree->Root->KeyIndex == 0)
376     {
377         //if root has no key
378         if(Tree->Root->Leaf == TRUE)
379         {
380             //if root is leaf, there are no keys in tree
381             Tree->Root = CreateBTreeNode();
382         }
383         else
384         {
385             //else, change root to its child
386             Tree->Root = Tree->Root->Childs[0];
387         }
388     }
389 }
391 int FindKey(BTreeNode *Node, int Key)
392 {
393     //find key equal or greater than Key
394     int Index = 0;
395     while(Index < Node->KeyIndex && Node->Keys[Index] < Key)
396     {
397         Index++;
398     }
399     return Index;
401 }
```

# B-Tree

```
404 void RemoveNode(BTreeNode *Node, int Key)
405 {
406     //find key index
407     int Index = FindKey(Node, Key);
408
409     //if key is exist in Node
410     if(Index < Node->KeyIndex && Node->Keys[Index] == Key)
411     {
412         if(Node->Leaf == TRUE)
413         {
414             RemoveFromLeaf(Node, Index);
415         }
416         else
417         {
418             RemoveFromNonLeaf(Node, Index);
419         }
420     }
421     else
422     {
423         if(Node->Leaf == TRUE)
424         {
425             //if Node is leaf and has no key, there are no key in tree
426             printf("Key %d is not exist in the tree\n", Key);
427             return;
428         }
429
430         //indicator which node has key to be removed
431         int Flag;
432         if(Node->KeyIndex == Index)
433         {
434             //key in childs[index - 1]
435             Flag = TRUE;
436         }
437         else
438         {
439             //key in childs[index]
440             Flag = FALSE;
441         }
442     }
443 }
```

# B-Tree

```
442
443     if(Node->Childs[Index]->KeyIndex < NODE_DEGREE)
444     {
445         //if the child where the key is supposed to exist has less key than NODE_DEGREE
446         //need to fill the child
447         Fill(Node, Index);
448     }
449
450     if(Flag && (Index > Node->KeyIndex))
451     {
452         //Key supposed in the childs[index - 1]
453         RemoveNode(Node->Childs[Index - 1], Key);
454     }
455     else
456     {
457         //Key supposed in the childs[index]
458         RemoveNode(Node->Childs[Index], Key);
459     }
460 }
461 }
```

# B-Tree

```
463 void RemoveFromLeaf(BTreeNode *Node, int Index)
464 {
465     //remove the index th key from node
466     for(int i = Index + 1; i < Node->KeyIndex; i++)
467     {
468         Node->Keys[i - 1] = Node->Keys[i];
469     }
470
471     Node->KeyIndex--;
472 }
```

# B-Tr

```
474 void RemoveFromNonLeaf(BTreeNode *Node, int Index)
475 {
476     int Key = Node->Keys[Index];
477
478     if(Node->Childs[Index]->KeyIndex > NODE_DEGREE - 1)
479     {
480         //if child[index] has at least NODE_DEGREE keys
481         //find predecessor and replace Keys[index] by predecessor-->remove Key
482         int Pred = RetPredecessor(Node, Index);
483         Node->Keys[Index] = Pred;
484         //remove predecessor in child[index]
485         RemoveNode(Node->Childs[Index], Pred);
486     }
487     else if (Node->Childs[Index + 1]->KeyIndex > NODE_DEGREE - 1)
488     {
489         //if child[index] has keys less than NODE_DEGREE,
490         //and if child[index + 1] has at least NODE_DEGREE keys
491         //need to replace Key with its successor
492         int Succ = RetSuccessor(Node, Index);
493         Node->Keys[Index] = Succ;
494         //remove successor in child[index + 1]
495         RemoveNode(Node->Childs[Index + 1], Succ);
496     }
497     else
498     {
499         //if both child[index] and child[index + 1] has less than NODE_DEGREE keys,
500         //need merge both child
501         Merge(Node, Index);
502         //after merge, Key is in the child[index], remove it
503         RemoveNode(Node->Childs[Index], Key);
504     }
505 }
506 }
```

# B-Tree

```
508 int RetPredecessor(BTreeNode *Node, int Index)
509 {
510     //find predecessor has maximum key
511     BTreeNode *Curr = Node->Childs[Index];
512     while(Curr->Leaf == FALSE)
513     {
514         Curr = Curr->Childs[Curr->KeyIndex];
515     }
516
517     return Curr->Keys[Curr->KeyIndex - 1];
518 }
519
520 int RetSuccessor(BTreeNode *Node, int Index)
521 {
522     //find successor has minimum key
523     BTreeNode *Curr = Node->Childs[Index + 1];
524     while(Curr->Leaf == FALSE)
525     {
526         Curr = Curr->Childs[0];
527     }
528
529     return Curr->Keys[0];
530 }
```



# B-Tree

```
532 void Fill(BTreeNode *Node, int Index)
533 {
534     if(Index != 0 && Node->Childs[Index - 1]->KeyIndex > NODE_DEGREE - 1)
535     {
536         //if the prev child has more than NODE_DEGREE - 1 keys, borrow key from that child
537         BorrowFromPrev(Node, Index);
538     }
539     else if(Index != Node->KeyIndex && Node->Childs[Index + 1]->KeyIndex > NODE_DEGREE - 1)
540     {
541         //if the next child has more than NODE_DEGREE - 1 keys, borrow key from that child
542         BorrowFromNext(Node, Index);
543     }
544     else
545     {
546         //merge with its sibling
547         if(Index != Node->KeyIndex)
548         {
549             //if child is not the last child, merge it with its next sibling
550             Merge(Node, Index);
551         }
552         else
553         {
554             //if child is the last child, merge it with its prev sibling
555             Merge(Node, Index - 1);
556         }
557     }
558 }
```

# B-Tre

```
560 void BorrowFromPrev(BTreeNode *Node, int Index)
561 {
562     BTreeNode *Child = Node->Childs[Index];
563     BTreeNode *Sibling = Node->Childs[Index - 1];
564
565     //moving all key in child[index] one step ahead
566     for(int i = Child->KeyIndex - 1; i > -1; i--)
567     {
568         Child->Keys[i + 1] = Child->Keys[i];
569     }
570
571     //if child[index] is not leaf, move all its child pointers one step ahead
572     if(Child->Leaf == FALSE)
573     {
574         for(int i = Child->KeyIndex; i > -1; i--)
575         {
576             Child->Childs[i + 1] = Child->Childs[i];
577         }
578     }
579     Child->Keys[0] = Node->Keys[Index - 1];
580
581     //moving last child of sibling as first child of child[index]
582     if(Child->Leaf == FALSE)
583     {
584         Child->Childs[0] = Sibling->Childs[Sibling->KeyIndex];
585     }
586     Node->Keys[Index - 1] = Sibling->Keys[Sibling->KeyIndex - 1];
587
588     Child->KeyIndex++;
589     Sibling->KeyIndex--;
590
591 }
```

# B-Tree

```
593 void BorrowFromNext(BTreeNode *Node, int Index)
594 {
595     BTreeNode *Child = Node->Childs[Index];
596     BTreeNode *Sibling = Node->Childs[Index + 1];
597
598     //keys[index] is inserted as the last key in childs[index]
599     Child->Keys[Child->KeyIndex] = Node->Keys[Index];
600
601     //first child of sibling is inserted as the last child into childs[index]
602     if(Child->Leaf == FALSE)
603     {
604         Child->Childs[Child->KeyIndex + 1] = Sibling->Childs[0];
605     }
606
607     //the first key from sibling is inserted into keys[index]
608     Node->Keys[Index] = Sibling->Keys[0];
609
610     //moving all keys in sibling one step behind
611     for(int i= 1; i < Sibling->KeyIndex; i++)
612     {
613         Sibling->Keys[i - 1] = Sibling->Keys[i];
614     }
615
616     //moving the child pointers one step behind
617     if(Sibling->Leaf == FALSE)
618     {
619         for(int i = 1; i < Sibling->KeyIndex; i++)
620         {
621             Sibling->Childs[i - 1] = Sibling->Childs[i];
622         }
623     }
624
625     Child->KeyIndex++;
626     Sibling->KeyIndex--;
627 }
```

```

629 void Merge(BTreeNode *Node, int Index)
630 {
631     BTreeNode *Child = Node->Childs[Index];
632     BTreeNode *Sibling = Node->Childs[Index + 1];
633
634     //moving key from the Node and insert it into NODE_DEGREE -1 th location of childs[index]
635     Child->Keys[NODE_DEGREE - 1] = Node->Keys[Index];
636
637     //copying the keys from childs[index + 1 ] to childs[index] at the end
638     for(int i = 0; i < Sibling->KeyIndex; i++)
639     {
640         Child->Keys[i + NODE_DEGREE] = Sibling->Keys[i];
641     }
642
643     //copying the child pointers from childs[index + 1] to childs[index]
644     if(Child->Leaf == FALSE)
645     {
646         for(int i = 0; i < Sibling->KeyIndex + 1; i++)
647         {
648             Child->Childs[i + NODE_DEGREE] = Sibling->Childs[i];
649         }
650     }
651
652     //moving all keys after index in the Node one step before to fill the space created by moving keys[index] to childs[index]
653     for(int i = Index + 1; i < Node->KeyIndex; i++)
654     {
655         Node->Keys[i - 1] = Node->Keys[i];
656     }
657
658     //moving the child pointers after index + 1 in the Node one step before
659     for(int i = Index + 2; i < Node->KeyIndex + 1; i++)
660     {
661         Node->Childs[i - 1] = Node->Childs[i];
662     }
663
664     Child->KeyIndex += Sibling->KeyIndex + 1;
665     Node->KeyIndex--;
666 }

```

# B-Tree

```
197 BTreeNode *CreateBTreeNode()
198 {
199     BTreeNode *temp = (BTreeNode*)malloc(sizeof(BTreeNode));
200
201     for(int i = 0; i < MAX_KEYS; i++)
202     {
203         temp->Keys[i] = 0;
204     }
205
206     for(int i = 0; i < MAX_CHILDS; i++)
207     {
208         temp->Childs[i] = NULL;
209     }
210
211     temp->KeyIndex = 0;
212     temp->Leaf = TRUE;
213
214     return temp;
215 }
216
217 BTree *CreateBTree()
218 {
219     BTree *Tree = (BTree*)malloc(sizeof(BTree));
220
221     BTreeNode *temp = CreateBTreeNode();
222
223     Tree->Degree = NODE_DEGREE; //may not used in this code
224     Tree->Root = temp;
225
226     return Tree;
227 }
```

# B-Tree

```
136 void PrintTree(BTreeNode *Node, int Level, int Blank) 157
137 { 158
138     //print b-tree as tree format 159
139     if(Node->KeyIndex == 0) 160
140     { 161
141         printf("Tree not exist\n"); 162
142         return; 163
143     } 164
144     if(Blank == TRUE) 165
145     { 166
146         for(int i = 0; i < Level; i++) 167
147         { 168
148             for(int j = 0; j < SPACE; j++) 169
149             { 170
150                 printf(" "); 171
151             } 172
152             printf("|"); 173
153         } 174
154     } 175
155 } 176
156
```

```
157 if(Node->KeyIndex > 0)
158 {
159     for(int i = 0; i < (SPACE - 4 * Node->KeyIndex); i++)
160     {
161         printf(" ");
162     }
163     printf("[");
164     for(int i = 0; i < Node->KeyIndex - 1; i++)
165     {
166         printf("%2d, ", Node->Keys[i]);
167     }
168     printf("%2d|", Node->Keys[Node->KeyIndex - 1]);
169 }
170 else
171 {
172     printf("[ ]");
173 }
174
```

# B-Tree

```
175
176     if(Node->Leaf == TRUE)
177     {
178         printf("\n");
179         return;
180     }
181     else
182     {
183         for(int i = 0; i < Node->KeyIndex + 1; i++)
184         {
185             if(i != 0)
186             {
187                 PrintTree(Node->Childs[i], Level + 1, TRUE);
188             }
189             else
190             {
191                 PrintTree(Node->Childs[i], Level + 1, FALSE);
192             }
193         }
194     }
195 }
```

# B-Tree

```
58 int main(int argc, char *argv[])
59 {
60
61     //targets of insert
62     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};
63
64     BTree *Tree = CreateBTree();
65     int i;
66
67     for(i=0; i<sizeof(test)/sizeof(int); i++)
68     {
69         printf("Insert : %d\n", test[i]);
70         Insert(Tree, test[i]);
71         PrintTree(Tree->Root, 0, FALSE);
72         printf("\n");
73     }
74
75     for(int i = 0; i < 3 * (SPACE + 1); i++)
76     {
77         printf("-");
78     }
79     printf("\n");
80 }
```



# B-Tree

```
81 //targets of delete
82 int testdel[] = {14, 3, 12, 27, 1, 10, 11, 25, 5, 9, 23, 20, 8, 4, 24, 1};
83
84 for(i = 0; i < sizeof(testdel)/sizeof(int); i++)
85 {
86     printf("Remove : %d\n", testdel[i]);
87     Remove(Tree, testdel[i]);
88     PrintTree(Tree->Root, 0, FALSE);
89     printf("\n");
90 }
91
92 for(int i = 0; i < 3 * (SPACE + 1); i++)
93 {
94     printf("-");
95 }
96 printf("\n");
97
```

# B-Tree

```
98      //search keys
99      int Target = 2;
100     printf("Search %d\n", Target);
101     Search(Tree->Root, Target);
102     printf("\n");
103
104     Target = 26;
105     printf("Search %d\n", Target);
106     Search(Tree->Root, Target);
107     printf("\n");
108
109     Target = 18;
110     printf("Search %d\n", Target);
111     Search(Tree->Root, Target);
112     printf("\n");
113
114     Target = 14;
115     printf("Search %d\n", Target);
116     Search(Tree->Root, Target);
117     printf("\n");
118     return 0;
119 }
```

# 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
```