HANYANG UNIVERSITY
DIVISION OF COMPUTER SCIENCE

Data Structures

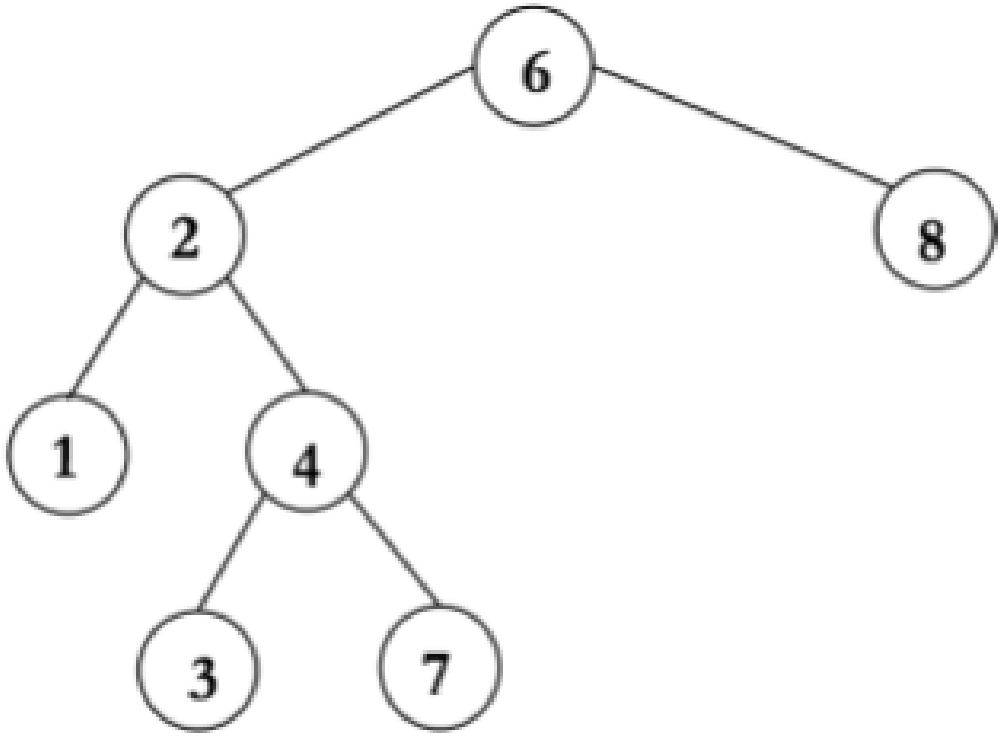
박영준 교수님

Lab5: BST

BST

- A rooted binary tree, whose internal nodes each store key(or value) and each have two distinguished sub-trees.
- The tree satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less then or equal to any key stored in the right sub-tree.

BST



BST ADT

Binary Tree

- BTNode *MakeBTNode(void);
  - Create & initialize binary tree node
  - Return binary tree node
- void SaveData(BTNode *Node, DATATYPE Data);
  - Save Data into Node
- DATATYPE RetData(BTNode *Node);
  - Return data of Node

BST ADT

Binary Tree

- void MakeSubTreeLeft(BTNode *Parent, BTNode *Child);
  - Link Child with left edge of Parent
- void MakeSubTreeRight(BTNode *Parent, BTNode *Child);
  - Link Child with right edge of Parent

BST ADT

Binary Tree

- BTNode *RetSubTreeLeft(BTNode *Node);
  - Return left child of Node
- BTNode *RetSubTreeRight(BTNode *Node);
  - Return right child of Node

BST ADT

Binary Tree

- BTNode *RemoveSubTreeLeft(BTNode *Node);
  - Remove left child of Node
  - Link parent and child of target node
  - Return target node, but not free target
- BTNode *RemoveSubTreeRight(BTNode *Node);
  - Remove right child of Node
  - Link parent and child of target node
  - Return target node, but no free target

BST ADT

Binary Tree

- void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child);
  - Change left child of Parent to Child
- void ChangeSubTreeRight(BTNode *Parent, BTNode *Child);
  - Change right child of Parent to Child

BST ADT

Binary Tree

- void PreorderTraversal(BTNode *Node);
- void InorderTraversal(BTNode *Node);
- void PostorderTraversal(BTNode *Node);

  - Traverse tree by given order

BST ADT

BST

- void MakeBST(BTNode **Node);
  - Initialize root node of BST
- void InsertBST(BTNode **Root, DATATYPE Data);
  - Insert new node with Data in Root BST
  - Compare each node and find location to inserted
    - If compared Node has bigger data, compare left child of Node
    - If compared Node has lower data, compare right child of Node
    - If there are no Node to compare, insert new node
  - Do not allow duplicate data

BST ADT

BST

- BTNode *SearchBST(BTNode *Node, DATATYPE Target);
  - Find node has Target
    - If compared Node has bigger data, compare left child of Node
    - If compared Node has lower data, compare right child of Node
  - If no node has Target, return NULL

BST ADT

BST

- BTNode *RemoveBST(BTNode **Root, DATATYPE Target);
  - Remove Node has Target in the BST
  - If Target is edge node, remove Target
  - If Target has single child, link parent and child of Target and remove Target
  - If Target has both child, replace Target with largest node in left sub-tree or smallest node in right sub-tree of Target
    And remove Target

BST ADT

BST

- void PrintAllBST(BTNode *Node)
  - Print all nodes of BST in given order

BST ADT

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

#define COUNT 12

typedef int DATATYPE;

typedef struct BTNode
{
    DATATYPE Data;
    struct BTNode *Left;
    struct BTNode *Right;
} BTNode;

//binary tree
BTNode *MakeBTNode(void);
DATATYPE RetData(BTNode *Node);
void SaveData(BTNode *Node, DATATYPE Data);

BTNode *RetSubTreeLeft(BTNode *Node);
BTNode *RetSubTreeRight(BTNode *Node);

void MakeSubTreeLeft(BTNode *Parent, BTNode *Child);
void MakeSubTreeRight(BTNode *Parent, BTNode *Child);

BTNode *RemoveSubTreeLeft(BTNode *Node);
BTNode *RemoveSubTreeRight(BTNode *Node);

void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child);
void ChangeSubTreeRight(BTNode *Parent, BTNode *Child);

//traversal
void PreorderTraversal(BTNode * Node);
void InorderTraversal(BTNode *Node);
void PostorderTraversal(BTNode *Node);

//BST
void MakeBST(BTNode **Node);

void InsertBST(BTNode **Root, DATATYPE Data);
BTNode *SearchBST(BTNode *Node, DATATYPE Target);
BTNode *RemoveBST(BTNode **Node, DATATYPE Target);

void PrintAllBST(BTNode *Node);

//util
void Print2D(BTNode *root, int space);
```

BST ADT

```
104 BTNode *MakeBTNode(void)
105 {
106     BTNode *Node = (BTNode*)malloc(sizeof(BTNode));
107     Node->Left = NULL;
108     Node->Right = NULL;
109     return Node;
110 }
111
112 DATATYPE RetData(BTNode *Node)
113 {
114     return Node->Data;
115 }
116
117 void SaveData(BTNode *Node, DATATYPE Data)
118 {
119     Node->Data = Data;
120 }
121
122 BTNode *RetSubTreeLeft(BTNode *Node)
123 {
124     return Node->Left;
125 }
126
127 BTNode *RetSubTreeRight(BTNode *Node)
128 {
129     return Node->Right;
130 }
```

```
132 void MakeSubTreeLeft(BTNode *Parent, BTNode *Child)
133 {
134     //if parent has child
135     if(Parent->Left != NULL)
136     {
137         free(Parent->Left);
138     }
139
140     Parent->Left = Child;
141 }
142
143 void MakeSubTreeRight(BTNode *Parent, BTNode *Child)
144 {
145     //if parent has child
146     if(Parent->Right != NULL)
147     {
148         free(Parent->Right);
149     }
150
151     Parent->Right = Child;
152 }
```

BST ADT

```
154 void PreorderTraversal(BTNode * Node)
155 {
156     if(Node == NULL)
157     {
158         return;
159     }
160
161     printf("%d ", Node->Data);
162     PreorderTraversal(Node->Left);
163     PreorderTraversal(Node->Right);
164 }
```

```
166 void InorderTraversal(BTNode *Node)
167 {
168     if(Node == NULL)
169     {
170         return;
171     }
172
173     InorderTraversal(Node->Left);
174     printf("%d ", Node->Data);
175     InorderTraversal(Node->Right);
176 }
177
178 void PostorderTraversal(BTNode *Node)
179 {
180     if(Node == NULL)
181     {
182         return;
183     }
184
185     PostorderTraversal(Node->Left);
186     PostorderTraversal(Node->Right);
187     printf("%d ", Node->Data);
188 }
```

BST ADT

```
190 BTNode *RemoveSubTreeLeft(BTNode *Node)
191 {
192     BTNode *Temp;
193
194     if(Node != NULL)
195     {
196         Temp = Node->Left;
197         Node->Left = NULL;
198     }
199
200     return Temp;
201 }
202
203 BTNode *RemoveSubTreeRight(BTNode *Node)
204 {
205     BTNode *Temp;
206
207     if(Node != NULL)
208     {
209         Temp = Node->Right;
210         Node->Right = NULL;
211     }
212
213     return Temp;
214 }
```

```
216 void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child)
217 {
218     Parent->Left = Child;
219 }
220
221 void ChangeSubTreeRight(BTNode *Parent, BTNode *Child)
222 {
223     Parent->Right = Child;
224 }
225
226 //BST
227 void MakeBST(BTNode **Node)
228 {
229     *Node = NULL;
230 }
```

BST ADT

```
232 void InsertBST(BTNode **Root, DATATYPE Data)
233 {
234     BTNode *Parent = NULL;
235     BTNode *Current = *Root;
236     BTNode *Temp = NULL;
237
238     //find where to add new temp node
239     while(Current != NULL)
240     {
241         //not allow duplicate data
242         if(Data == RetData(Current))
243         {
244             return;
245         }
246
247         Parent = Current;
248
249         if(RetData(Current) > Data)
250         {
251             Current = RetSubTreeLeft(Current);
252         }
253         else
254         {
255             Current = RetSubTreeRight(Current);
256         }
257     }
```

```
258
259     //create new temp node
260     Temp = MakeBTNode();
261     SaveData(Temp, Data);
262
263     //add new node on the sub of parent
264     if(Parent != NULL)
265     {
266         //if new temp is not root
267         if(Data < RetData(Parent))
268         {
269             MakeSubTreeLeft(Parent, Temp);
270         }
271         else
272         {
273             MakeSubTreeRight(Parent, Temp);
274         }
275     }
276     else
277     {
278         //if new temp is root
279         *Root = Temp;
280     }
281 }
```

BST ADT

```
283 BTNode *SearchBST(BTNode *Node, DATATYPE Target)
284 {
285     BTNode *Current = Node;
286     DATATYPE Data;
287 
288     while(Current != NULL)
289     {
290         Data = RetData(Current);
291 
292         if(Target == Data)
293         {
294             return Current;
295         }
296         else if(Target < Data)
297         {
298             Current = RetSubTreeLeft(Current);
299         }
300         else
301         {
302             Current = RetSubTreeRight(Current);
303         }
304 
305     }
306     return NULL;
307 }
```

BST ADT

```
309 BTNode *RemoveBST(BTNode **Root, DATATYPE Target)
310 {
311     //create virtual root
312     BTNode *VirtualRoot = MakeBTNode();
313
314     BTNode *Parent = VirtualRoot;
315     BTNode *Current = *Root;
316     BTNode *TargetNode;
317
318     // make root node to be right child of virtual root
319     ChangeSubTreeRight(VirtualRoot, *Root);
320
321     //search target node
322     while(Current != NULL && RetData(Current) != Target)
323     {
324         Parent = Current;
325
326         if(Target < RetData(Current))
327         {
328             Current = RetSubTreeLeft(Current);
329
330         }
331         else
332         {
333             Current = RetSubTreeRight(Current);
334         }
335     }
```

```
336
337     //if target not exist
338     if(Current == NULL)
339     {
340         return NULL;
341     }
342
343     TargetNode = Current;
344
345     //if target node is edge node
346     if(RetSubTreeLeft(TargetNode) == NULL && RetSubTreeRight(TargetNode) == NULL)
347     {
348         if(RetSubTreeLeft(Parent) == TargetNode)
349         {
350             RemoveSubTreeLeft(Parent);
351         }
352         else
353         {
354             RemoveSubTreeRight(Parent);
355         }
356     }
```

BST ADT

```
357     else if(RetSubTreeLeft(TargetNode) == NULL || RetSubTreeRight(TargetNode) == NULL)
358     {
359         //if target has single child
360         BTNode *ChildofTarget;
361
362         //find target
363         if(RetSubTreeLeft(TargetNode) != NULL)
364         {
365             ChildofTarget = RetSubTreeLeft(TargetNode);
366         }
367         else
368         {
369             ChildofTarget = RetSubTreeRight(TargetNode);
370         }
371
372         //link parent and child of target
373         if(RetSubTreeLeft(Parent) == TargetNode)
374         {
375             ChangeSubTreeLeft(Parent, ChildofTarget);
376         }
377         else
378         {
379             ChangeSubTreeRight(Parent, ChildofTarget);
380         }
381     }
```

BST ADT

```
382     else
383     {
384         //if target has both child
385         BTNode *MinimumNode = RetSubTreeRight(TargetNode);
386         BTNode *ParentofMinimum = TargetNode;
387
388         DATATYPE Backup;
389
390         //find node to replace target node
391         while(RetSubTreeLeft(MinimumNode) != NULL)
392         {
393             ParentofMinimum = MinimumNode;
394             MinimumNode = RetSubTreeLeft(MinimumNode);
395         }
396
397         //backup target data
398         Backup = RetData(TargetNode);
399         //replace data of target node
400         SaveData(TargetNode, RetData(MinimumNode));
401
402         //link parent and child of MinimumNode
403         if(RetSubTreeLeft(ParentofMinimum) == MinimumNode)
404         {
405             ChangeSubTreeLeft(ParentofMinimum, RetSubTreeRight(MinimumNode));
406         }
407         else
408         {
409             ChangeSubTreeRight(ParentofMinimum, RetSubTreeRight(MinimumNode));
410         }
411
412         TargetNode = MinimumNode;
413         SaveData(TargetNode, Backup);
414     }
```

```
415
416     //if target node is root
417     if(RetSubTreeRight(VirtualRoot) != *Root)
418     {
419         *Root = RetSubTreeRight(VirtualRoot);
420     }
421
422     free(VirtualRoot);
423     return TargetNode;
424 }
```

BST ADT

```
426 void PrintAllBST(BTNode *Node)
427 {
428     //PreorderTraversal(Node);
429 //  InorderTraversal(Node);
430     //PostorderTraversal(Node);
431     Print2D(Node, 0);
432 }
```

```
434 void Print2D(BTNode *root, int space)
435 {
436     if(root == NULL)
437     {
438         return;
439     }
440
441     space += COUNT;
442
443     Print2D(root->Right, space);
444
445     printf("\n");
446     for(int i = COUNT; i < space; i++)
447     {
448         printf(" ");
449     }
450     printf("%d\n", root->Data);
451
452     Print2D(root->Left, space);
453 }
```

Lab5: BST

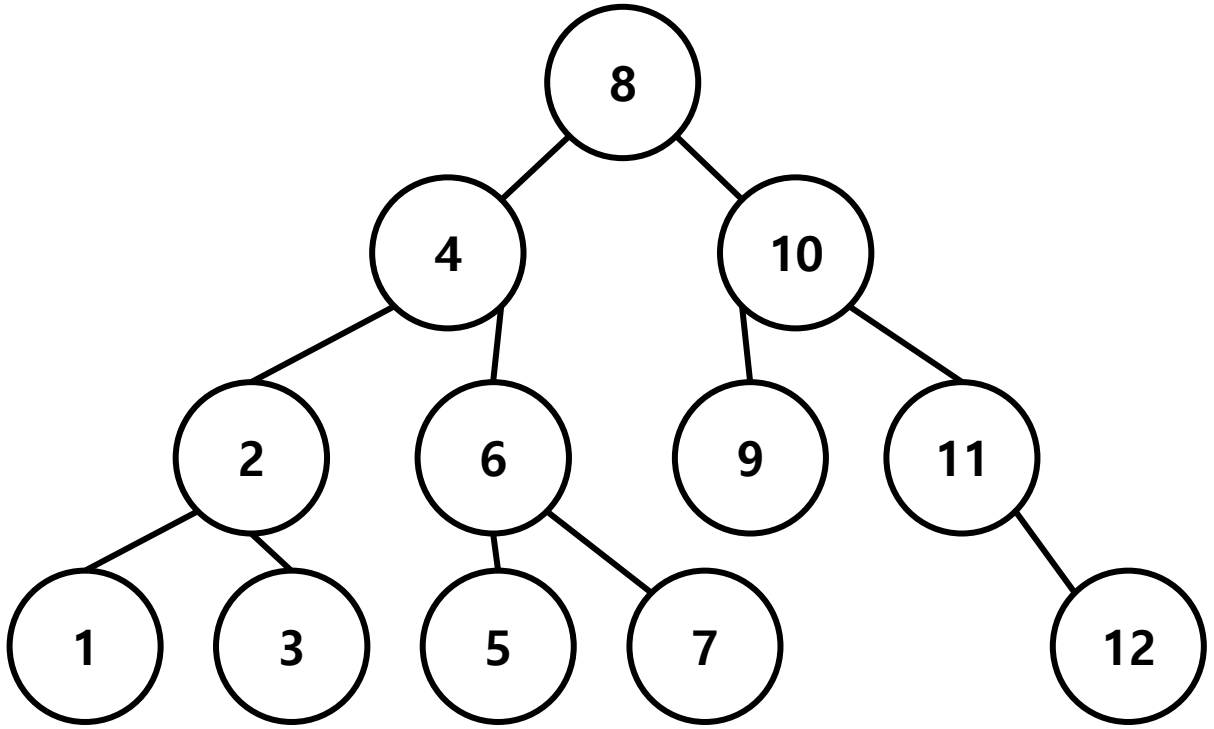
- Submit on GitLab
- BST replace with maximum value in remove operation
- Create Lab5 directory on your own GitLab project
- Submit file : source_code(c only, run on linux)
- Filename : StudentID_lab5.c
- Input file : no

Lab5: BST

- BST replace with maximum value in remove operation
- Change replace algorithm in RemoveBST()
- Node with the minimum value among subtree of the right child
- → Node with the maximum value among subtree of the left child

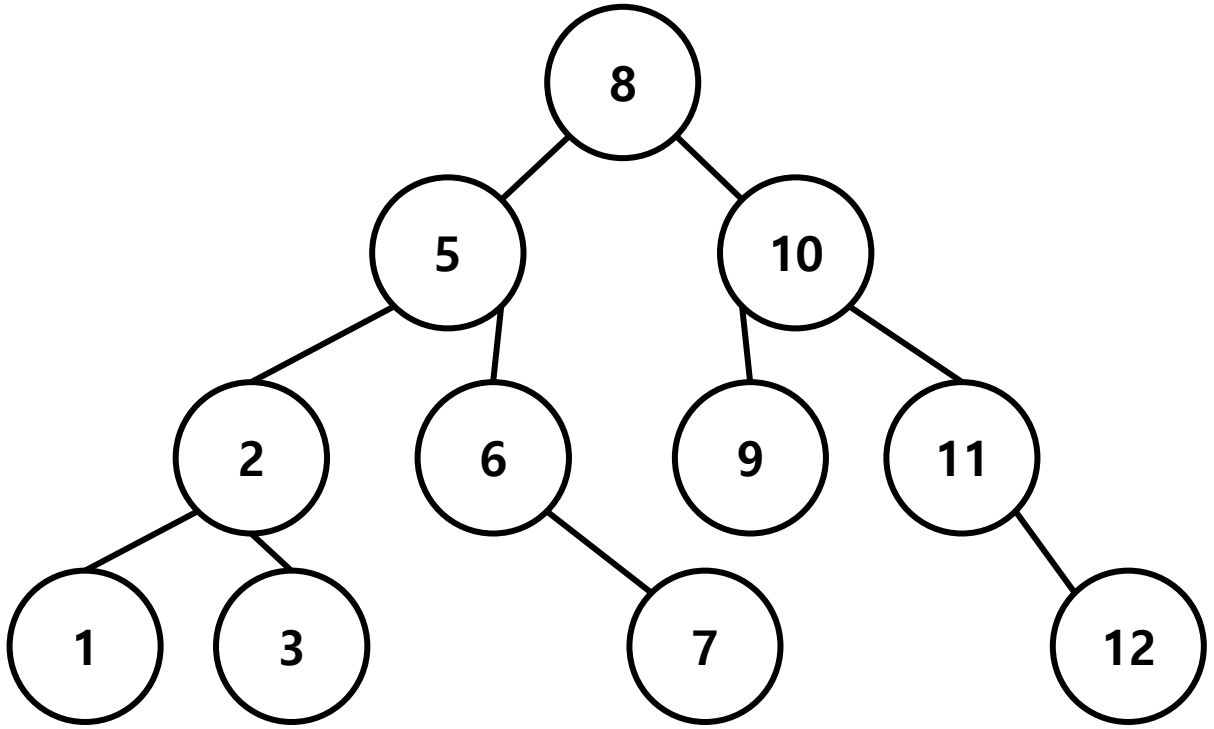
Lab5: BST

- Example
  - In original code, delete 4
  - Find minimum value in the right subtree
  - → 5
  - Replace 4 with 5
  - And delete 4



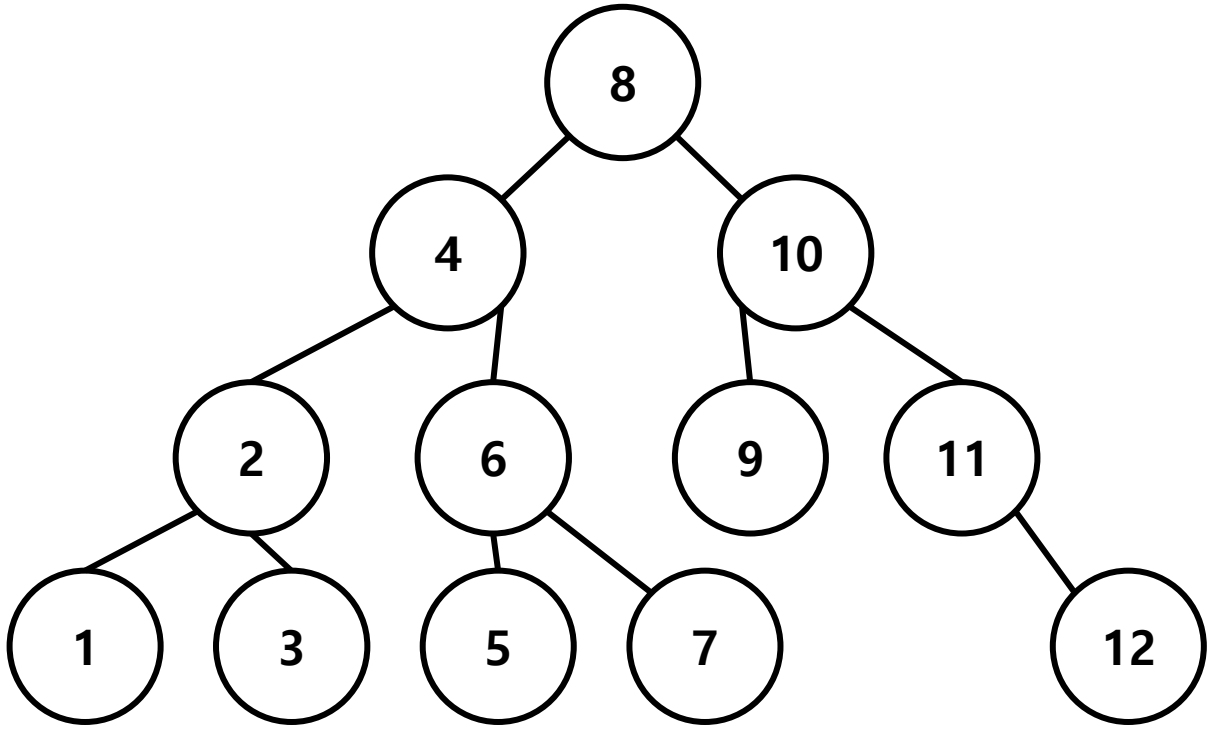
Lab5: BST

- Example
  - In original code, delete 4
  - Find minimum value in the right subtree
  - → 5
  - Replace 4 with 5
  - And delete 4



Lab5: BST

- Example
  - Change to
  - When delete 4
  - Find maximum value in the left subtree
  - → 3
  - Replace 4 with 3
  - And delete 4



Lab5: BST

- Example
  - Change to
  - When delete 4
  - Find maximum value in the left subtree
  - → 3
  - Replace 4 with 3
  - And delete 4