Data Structure

실습 9

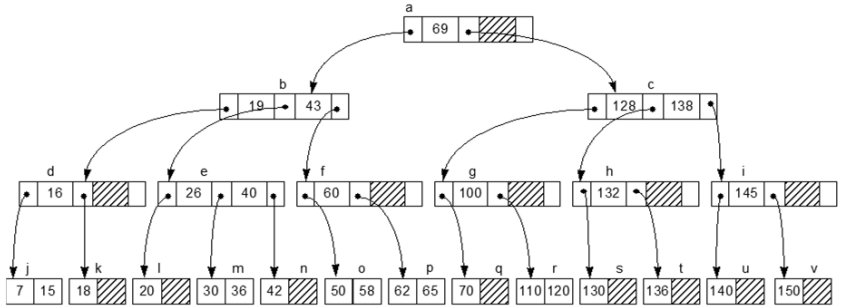
0. 이번 주 실습 내용

- **B-Tree**
  - B-Tree의 정의
  - B-Tree의 응용 사례
  - B-Tree data Insertion & Deletion
- **B-Tree 실습**
  - B-Tree data insertion 구현 실습

1. B-Tree

- **B-Tree (정의): M-way Search Tree + AVL Tree**
  - **M-way Search Tree** : 자식 노드가 최대 m개인 탐색 트리 (m≥3)
    - 2-way Search Tree == Binary Search Tree

  - 트리의 root 노드는 최소한 2개의 sub tree를 가진다.
  - root와 leaf노드를 제외한 트리의 각 노드들은 최소한 $\lceil m/2 \rceil$개의 sub tree를 가진다.
  - 트리의 모든 leaf 노드들은 같은 level에 존재하며 최소한 $\lceil m/2 \rceil - 1$개의 키 값을 가진다.



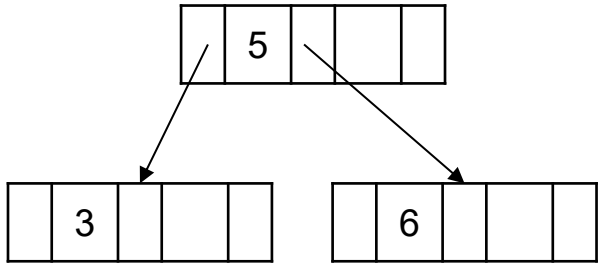
m=3인 B-Tree

1. B-Tree

- **B-Tree의 특성**
  - 트리의 root 노드는 최소한 2개의 sub tree를 가진다.
    → 최소 1개의 키 값 필요
  - root와 leaf노드를 제외한 트리의 각 노드들은 최소한 $\lceil m/2 \rceil$개의 sub tree를 가진다.
    → 최소 $\lceil m/2 \rceil$ - 1개의 키 값 필요
  - 트리의 모든 leaf 노드들은 같은 level에 존재하며 최소한 $\lceil m/2 \rceil - 1$개의 키 값을 가진다.

m=3인 B-Tree

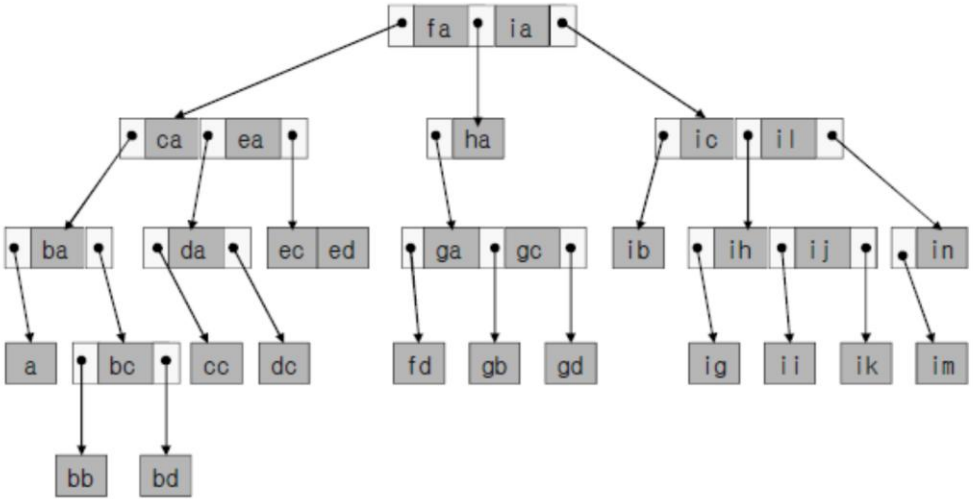
Root == Leaf인 경우

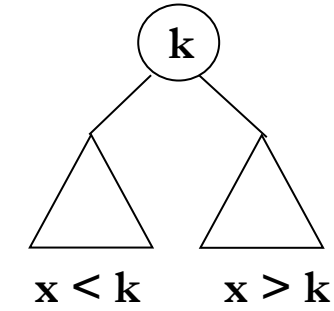


모든 노드가 $\lceil m/2 \rceil$ - 1개의 키를 보유

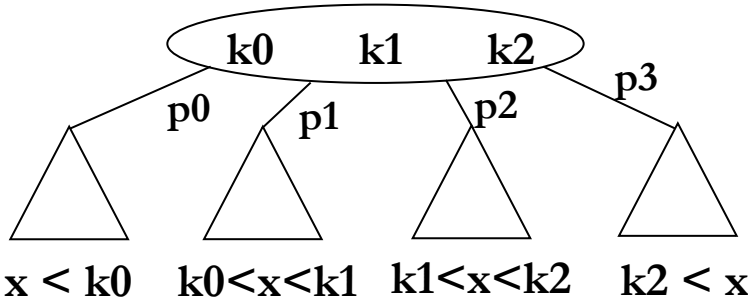
1. B-Tree

- **M-way Search Tree**
  - **Example (m=3)**





2-way Search Tree
(Binary Search Tree)



4-way Search Tree

1. B-Tree

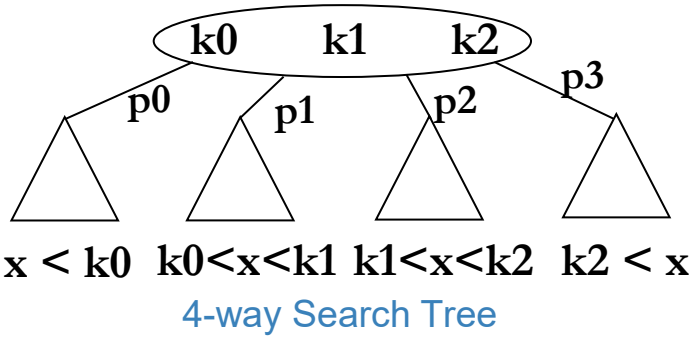
- **M-way Search Tree**
  - **구조**

n	P_0	k_0	P_1	k_1	…	P_{n-1}	k_{n-1}	P_n

- P_0, P_1 , …,P_n은 노드의 sub tree에 대한 포인터
- k_0, …, k_{n-1}은 키 값들
- n : 키의 개수로 포인터의 수보다 1만큼 적다
  m-way search tree이므로 n ≤ m-1 이 성립한다.

1. B-Tree

- **M-way Search Tree**
  - **조건**

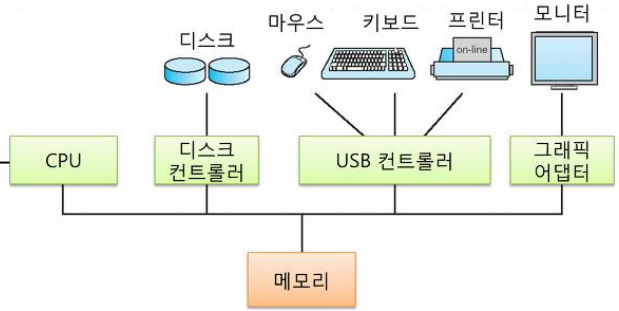


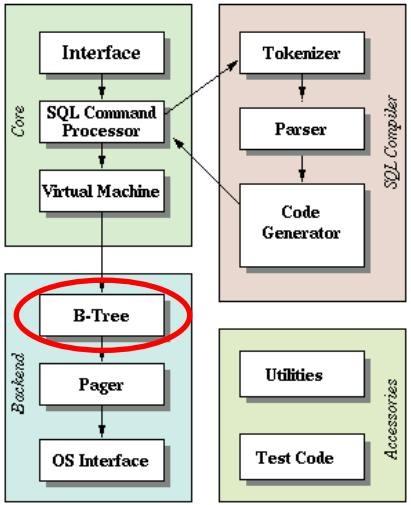
4-way Search Tree

  - 노트의 키 값은 오름차순
    - i = 0, …, n-2인 i 에 대해 $k_i < k_{i+1}$ 를 만족
  - i= 0, …, n-1인 i에 대해 P_i가 가리키는 sub tree의 모든 키 값은 k_i의 값보다 작음
  - P_n가 가리키는 sub tree의 모든 노드들의 키 값은 k_{n-1}의 값보다 큼
  - i= 0, …, n인 i에 대해 P_i가 가리키는 sub tree도 m-way search tree임

1. B-Tree

- **어디에서 쓸까? (실생활 응용)**
  - Database and File System





SQLite

1. 디스크에 저장된 방대한 data를 한꺼번에 메모리에 올리는 것은 불가능.
2. data들을 쉽게 찾을 수 있는 Index 구조의 새 data가 필요.

➔ 메모리에 Index data를 올려서 원하는 data가 저장된 디스크 위치를 찾아내고 해당 data만 메모리 상으로 올려서 작업 수행이 가능

1. 디스크에서 메모리상으로 data를 올릴 때 block 단위로(일정 크기)수행
2. 따라서 Index data 구조가 BST일 경우 노드 하나 올리는 데 나머지 빈 공간들이 낭비됨

➔ 노드의 크기가 block 크기 정도로 조절된 B-Tree를 사용하면 매우 효율적이다!

1. B-Tree

- **Data Insertion**

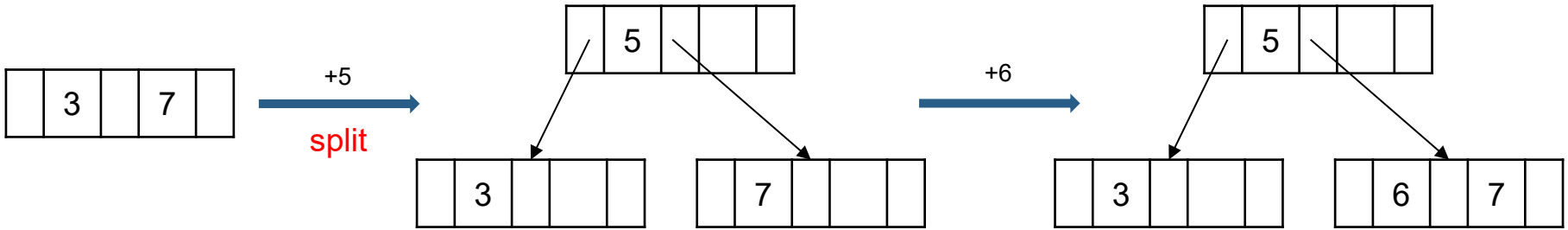
  Step1: Data는 항상 Leaf 노드에 추가된다 (BST와 같은 방식)

  Step2: 추가될 Leaf 노드에 data가 가득 차 있지 않은 경우 (키의 개수 < m-1)

  - 그냥 Leaf 노드에 data 추가

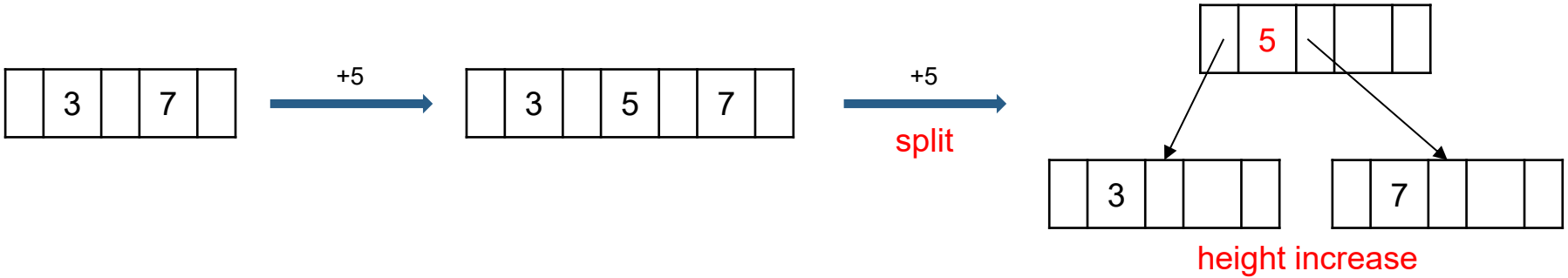
  Step3: 추가될 Leaf 노드에 data가 가득 차 있는 경우 (키의 개수 = m-1)

  - Leaf 노드 '**Node Split**' 수행
  - '**Node Split**'이후 해당 Leaf 노드의 parent 노드가 가득 차게 되면 또 다시 '**Node Split**' 수행



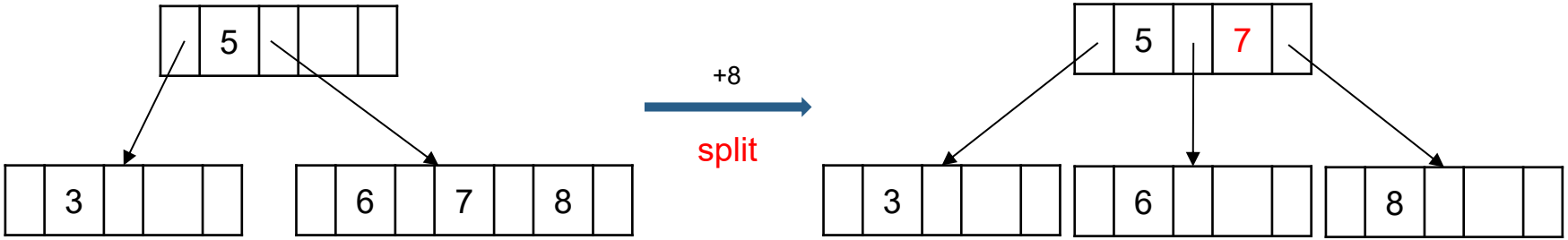
1. B-Tree

- **Data Insertion** (insert 5)
  - **Node Split**
    1. Overflow된 Leaf 노드에서, 중앙값을 기준으로 2개의 노드로 분할
    2. 중간에 위치한 키 값을 상위 노드(parent)로 이동
    3. Parent 노드에 위치한 키 값의 오른쪽 sub tree에 분할된 Leaf 노드를 연결
    4. Parent 노드에서 overflow가 발생했다면 recursive하게 parent에서 1부터 반복
       → root에서 overflow가 발생하면 트리 높이 증가



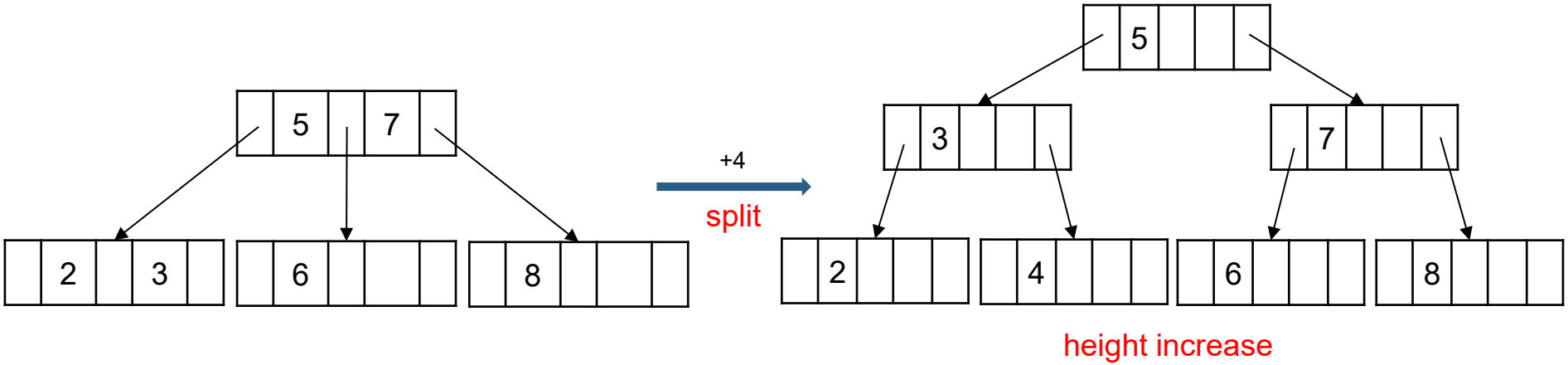
1. B-Tree

- **Data Insertion** (insert 8)
  - **Node Split**
    1. Overflow된 Leaf 노드에서, 중앙값을 기준으로 2개의 노드로 분할
    2. 중간에 위치한 키 값을 상위 노드(parent)로 이동
    3. Parent 노드에 위치한 키 값의 오른쪽 sub tree에 분할된 Leaf 노드를 연결
    4. Parent 노드에서 overflow가 발생했다면 recursive하게 parent에서 1부터 반복
       → root에서 overflow가 발생하면 트리 높이 증가



1. B-Tree

- **Data Insertion** (insert 2,4)
  - **Node Split**
    1. Overflow된 Leaf 노드에서, 중앙값을 기준으로 2개의 노드로 분할
    2. 중간에 위치한 키 값을 상위 노드(parent)로 이동
    3. Parent 노드에 위치한 키 값의 오른쪽 sub tree에 분할된 Leaf 노드를 연결
    4. Parent 노드에서 overflow가 발생했다면 recursive하게 parent에서 1부터 반복
       → root에서 overflow가 발생하면 트리 높이 증가



1. B-Tree

- **Data Deletion**

  Step1: 지우고자 하는 Data가 존재하는 노드를 탐색

  Step2: Leaf 노드가 아니라면 해당 Data를 대체할 키 값을 찾는다 (BST와 같은 방식으로 successor를 찾아 교체)

  Step3: Data 삭제 이후 해당 노드에 존재하는 키의 개수가 $\lceil m/2 \rceil - 1$개 이상인 경우

  - B-Tree를 유지하고 있으므로 연산 종료

  Step4: Data 삭제 이후 해당 노드에 존재하는 키의 개수가 $\lceil m/2 \rceil - 1$개 보다 작을 경우 '**Key-Rotation**'이나 '**Merge**'를 수행한다

  - '**Key-Rotation**' : 해당 노드의 sibling 노드로부터 한 개의 키 값을 차출(키의 개수가 충분한 경우)하여 parent 노드로 이동시키고 parent 노드의 키 값을 해당 노드로 이동

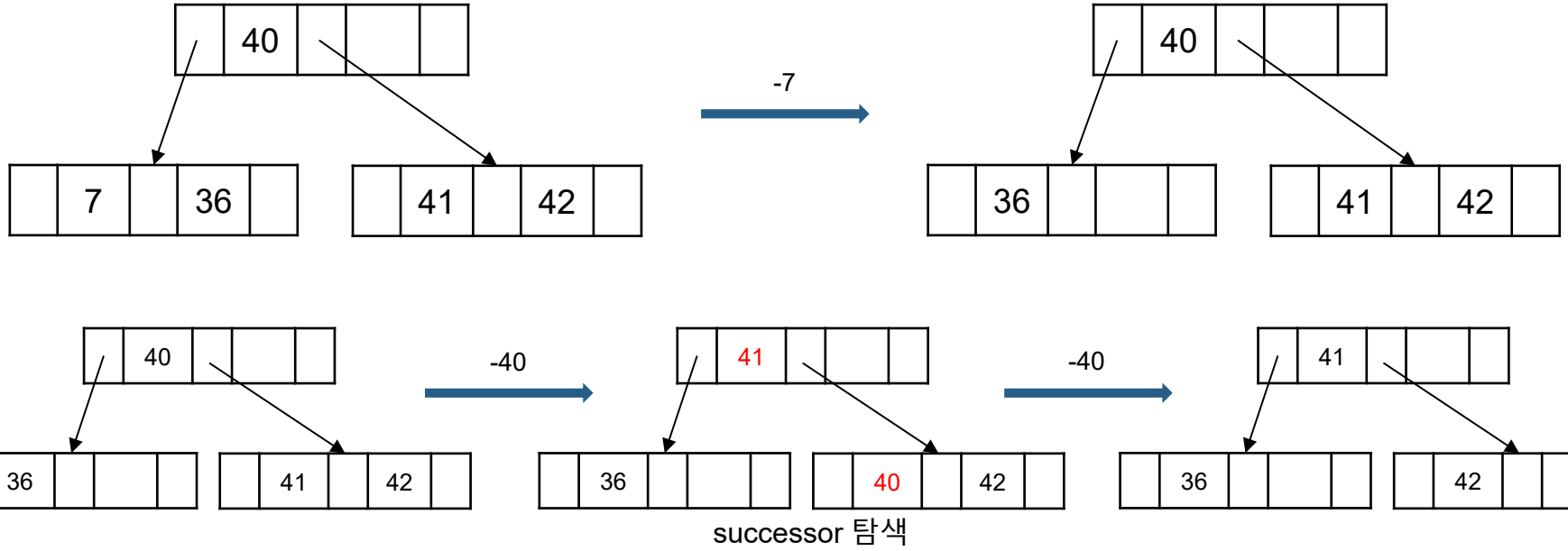
  - '**Merge(Join)**' : Key-Rotation이 불가능한 경우 해당 노드의 sibling 노드의 키 값들과 parent 노드의 키 값들을 모아 하나의 노드로 합병. 이 때 합병되는 노드의 parent 노드에 대해 다시 삭제 수행 (Go to 3)

1. B-Tree

- **Data Deletion** (delete 7,40)

  Step3: Data 삭제 이후 해당 노드에 존재하는 키의 개수가 $\lceil m/2 \rceil - 1$개 이상인 경우

  - B-Tree를 유지하고 있으므로 연산 종료

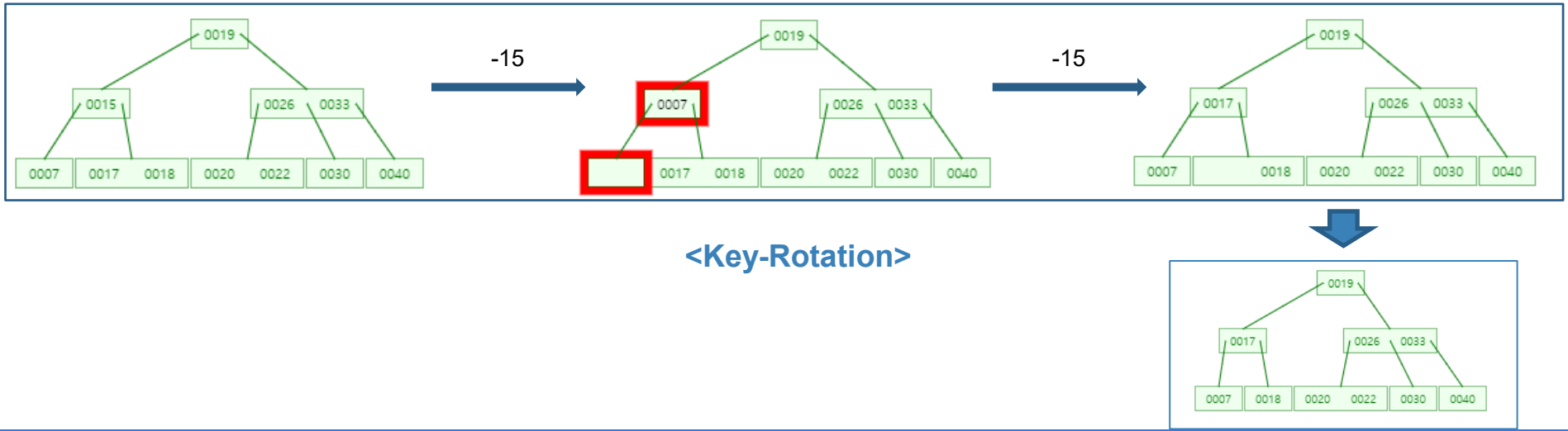


1. B-Tree

- **Data Deletion** (delete 15)

  Step4: Data 삭제 이후 해당 노드에 존재하는 키의 개수가 $\lceil m/2 \rceil - 1$개 보다 작을 경우 '**Key-Rotation**'이나 '**Merge**'를 수행한다

  - '**Key-Rotation**' : 해당 노드의 sibling 노드로부터 한 개의 키 값을 차출(키의 개수가 충분한 경우)하여 parent 노드로 이동시키고 parent 노드의 키 값을 해당 노드로 이동

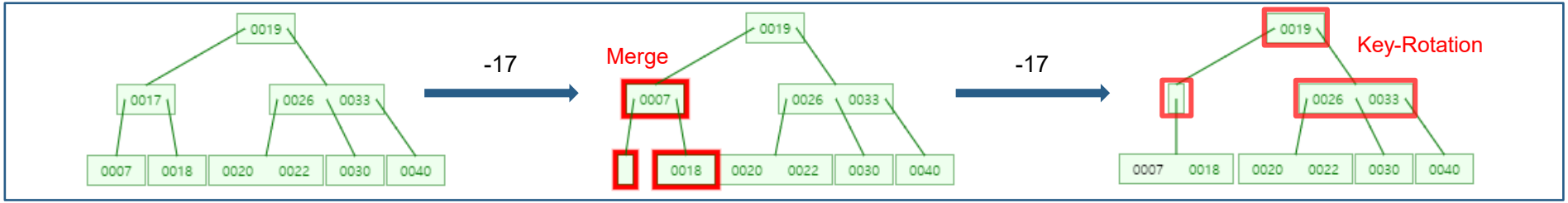


1. B-Tree

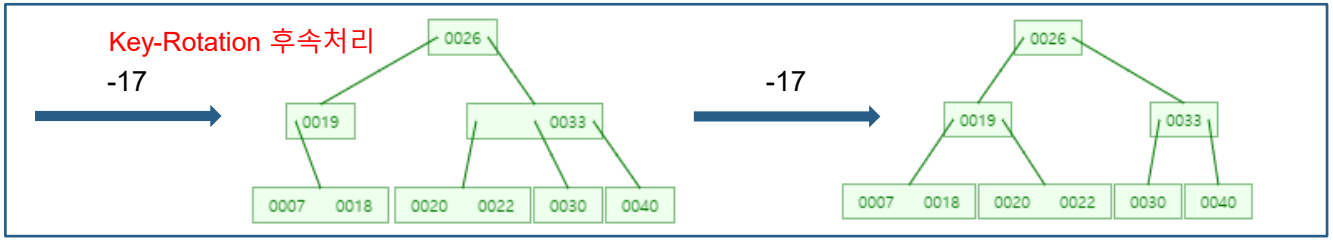
- **Data Deletion** (delete 17)

  Step4: Data 삭제 이후 해당 노드에 존재하는 키의 개수가 $\lceil m/2 \rceil - 1$개 보다 작을 경우 '**Key-Rotation**'이나 '**Merge**'를 수행한다

  - '**Merge(Join)**' : Key-Rotation이 불가능한 경우 해당 노드의 sibling 노드의 키 값들과 parent 노드의 키 값들을 모아 하나의 노드로 합병. 이 때 합병되는 노드의 parent 노드에 대해 다시 삭제 수행 (Go to 3)



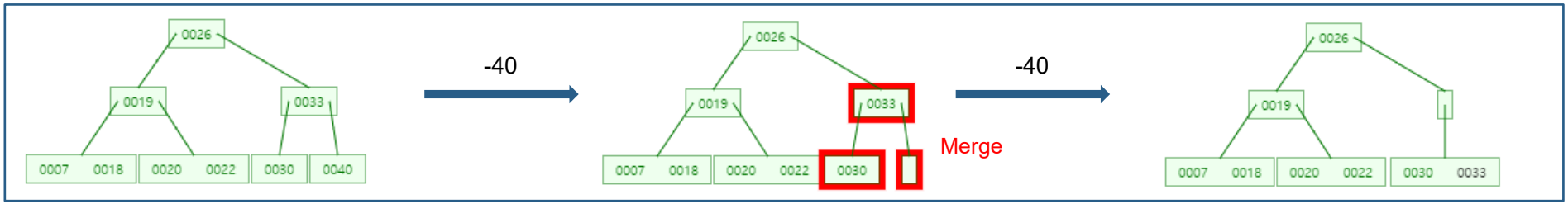
<Merge>

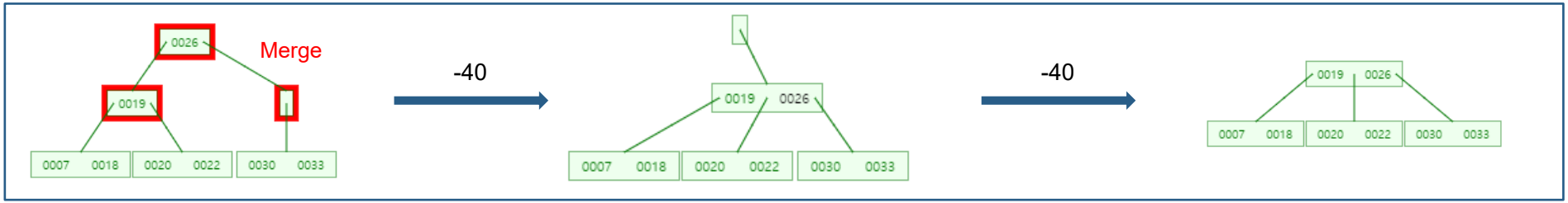


<parent 노드에 대해 다시 삭제 수행 (Go to 3)>

1. B-Tree

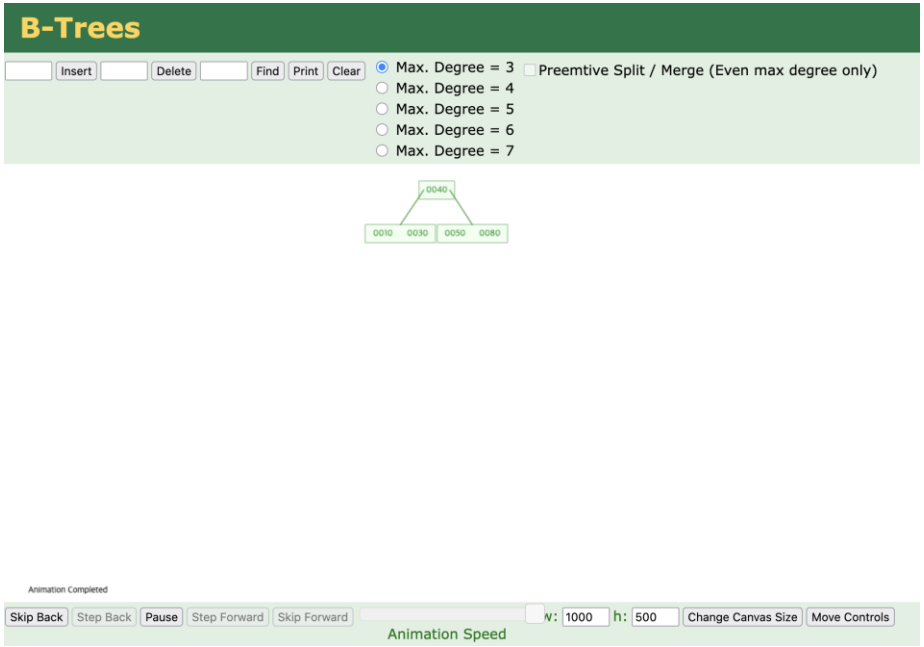
- **Data Deletion** (delete 40)





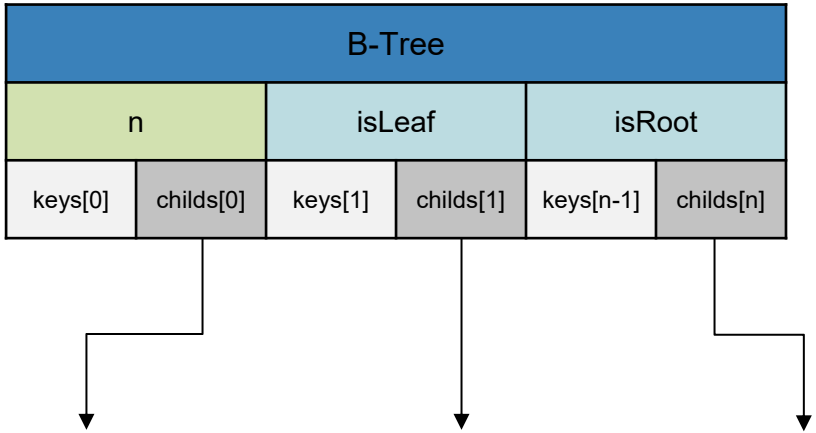
1. B-Tree

- Insertion & Deletion Exercise (B-Tree Visualization)
  - B-Tree 시각화 사이트: https://www.cs.usfca.edu/~galles/visualization/BTree.html

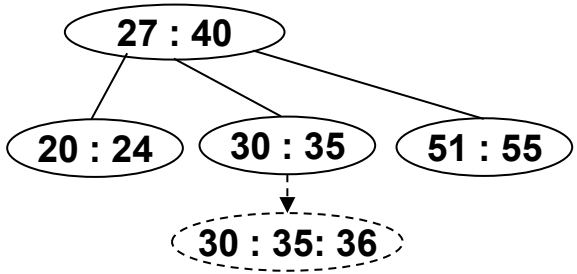


2. B-Tree – 실습

- **B-Tree Data Type**




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

#define M_WAY 3

typedef struct BTNode {
    int n;
    int isLeaf;          //leaf node인 경우 1
    int isRoot;          //root node인 경우 1
    int keys[M_WAY];     //3-Way B-Tree이기 때문에 최대 2개의 키값을 갖지만
                         //split 을 용이하게 하기 위해 1개의 여유 키값을 갖도록 선언
    struct BTNode* childs[M_WAY+1]; //child node pointer의 개수도 같은 이유로 +1
}BTNode;

BTNode* initBTNode();
BTNode* BTInsert(BTNode* root, int key);
BTNode* splitChild(BTNode* root);
void inorderTraversal(BTNode * root);
```



insert(36)

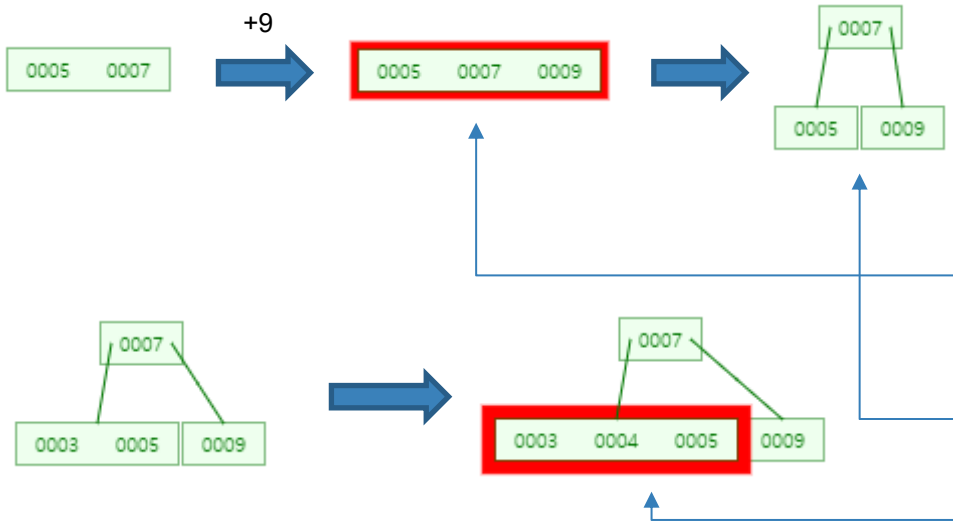
2. B-Tree – 실습

- **Create B-tree node** (BTNode* initBTNode())
  - **leaf 노드 하나 생성 + 초기화**

```
16  BTNode* initBTNode()
17  {
18      int i;
19      BTNode* newBTNode;
20      newBTNode = (BTNode*)malloc(sizeof(BTNode));
21      newBTNode->n = 0;
22      newBTNode->isLeaf = 1;      //항상 leaf node에서 생기므로
23      newBTNode->isRoot = 0;
24
25      for (i = 0; i < M_WAY+1; i++)
26          newBTNode->childs[i] = NULL;
27
28      return newBTNode;
29  }
```

2. B-Tree – 실습

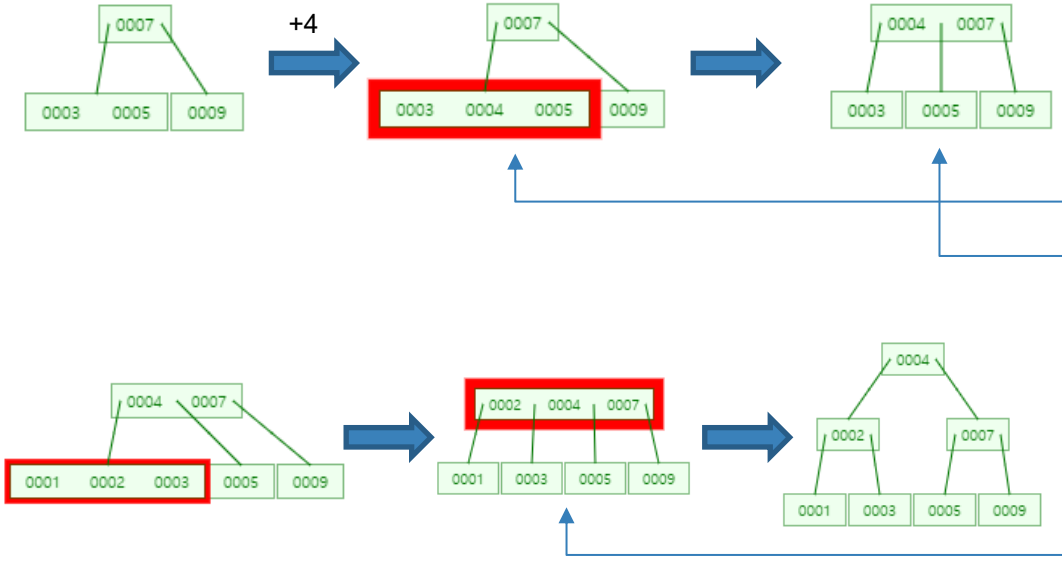
- **Insert Operation** (`BTNode* BTInsert(BTNode* root, int key)`)




```
31  BTNode* BTInsert(BTNode* root, int key)
32  {
33      int i,pos,mid;
34      BTNode* split;
35
36
37      //해당 노드가 leaf 노드일 때
38      if(root->isLeaf)
39      {
40          //노드의 키 값들 중 key가 들어갈 위치 탐색
41          for (pos = 0; pos < root->n; pos++)
42              if (root->keys[pos] > key)
43                  break;
44
45          //정렬을 통해 위치를 맞춰주고 pos 위치에 key를 삽입
46          for (i = root->n; i > pos; i--)
47
48
49
50
51          //insertion 수행 후 노드의 키 값의 개수가 다 찼을 때
52          //해당 노드가 root 노드인 경우에만 split 수행
53          if
54              return splitChild(root);
55          //그냥 leaf 노드인 경우 상위 함수에서 split 수행
56          return root;
57
58      }
```

2. B-Tree – 실습

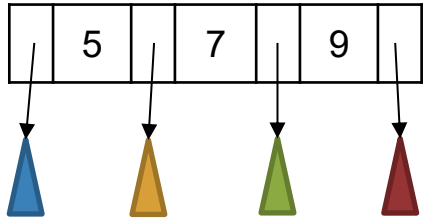
- **Insert Operation** (BTNode* BTInsert(BTNode* root, int key))



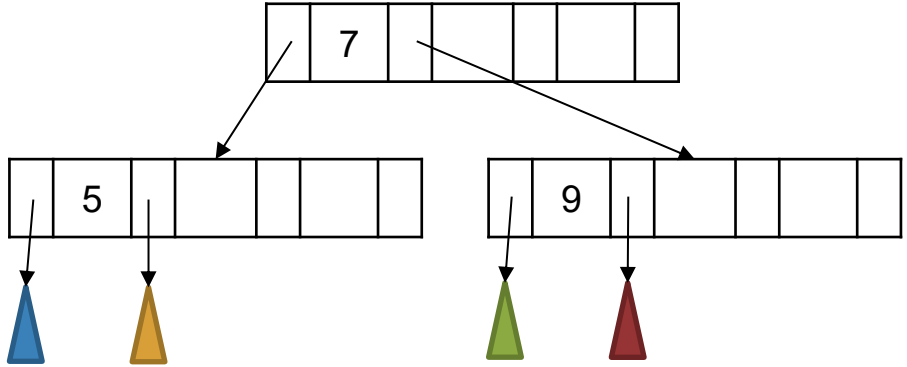

```
59      //해당 노드가 leaf 노드가 아닐 때
60      else
61      {
62          //노드의 키 값들 중 key가 들어갈 위치 탐색
63          for (pos = 0; pos < root->n; pos++)
64              if (root->keys[pos] > key)
65                  break;
66          //해당 위치에 존재하는 child 노드로 들어가 leaf 노드까지 탐색
67
68
69          //insertion 수행 후 child 노드의 키의 개수가 다 찼을 때
70          if (
71          {
72              //child 노드에 대해 split 수행
73              split = splitChild(root->childs[pos]);
74              //split 수행 후 적절하게 parent 노드에 추가
75              for (i = root->n; i > pos; i--)
76              {
77
78
79              }
80
81
82
83
84          }
85          //최종적으로 root 노드가 실제 b-tree의 root 노드인 경우 중에
86          //키의 개수가 다 찼을 때 split 수행
87          if (
88              return
89
90          return root;
91      }
92  }
```

2. B-Tree – 실습

- **Split Operation** (`BTNode* splitChild(BTNode* root)`)
  - 노드에 M_WAY 개수 만큼의 key 값이 들어 있을 때
  1. 새로운 Parent 노드를 생성하고 가운데 key 값을 전달
  2. 새로운 Sibling 노드를 생성하고 key 값을 분할
  3. sub tree에 해당하는 child 노드들도 분할
  4. Parent 노드에 알맞게 연결

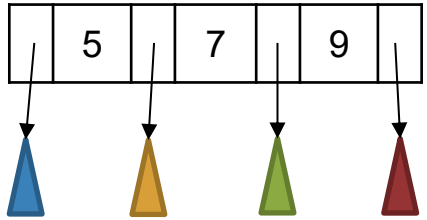



```
94   BTNode* splitChild(BTNode* root)
95   {
96       int i, mid;
97       //root 노드를 split 하게 되면
98       //      newParent
99       //  root        newSibling
100      //형태로 split이 일어나게끔 수행
101      BTNode* newParent;
102      BTNode* newSibling;
103
104      newParent = initBTNode();
105      newParent->isLeaf = 0;
106      //root 노드가 실제 root 노드였다면
107      //newParent 노드가 실제 root 노드가 된다
108      if (root->isRoot)
109
110
111
112      //newSibling 노드는 root와 같은 level 상에 존재한다
113      newSibling = initBTNode();
114
115
```

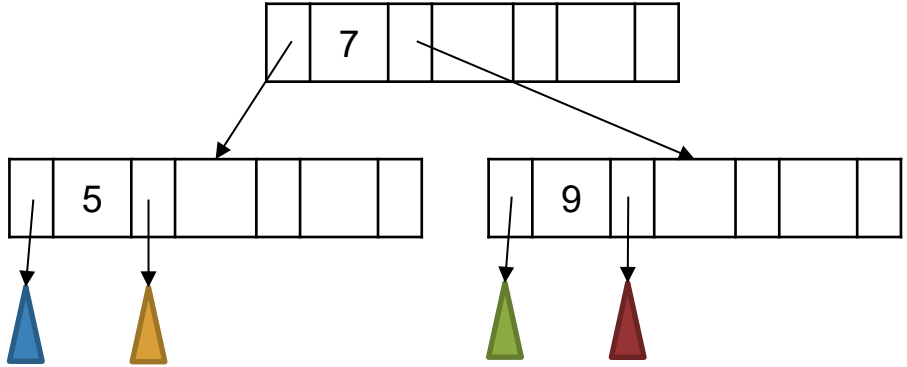


2. B-Tree – 실습

- **Split Operation** (`BTNode* splitChild(BTNode* root)`)
  - 노드에 M_WAY 개수 만큼의 key 값이 들어 있을 때
  1. 새로운 Parent 노드를 생성하고 가운데 key 값을 전달
  2. 새로운 Sibling 노드를 생성하고 key 값을 분할
  3. sub tree에 해당하는 child 노드들도 분할
  4. Parent 노드에 알맞게 연결

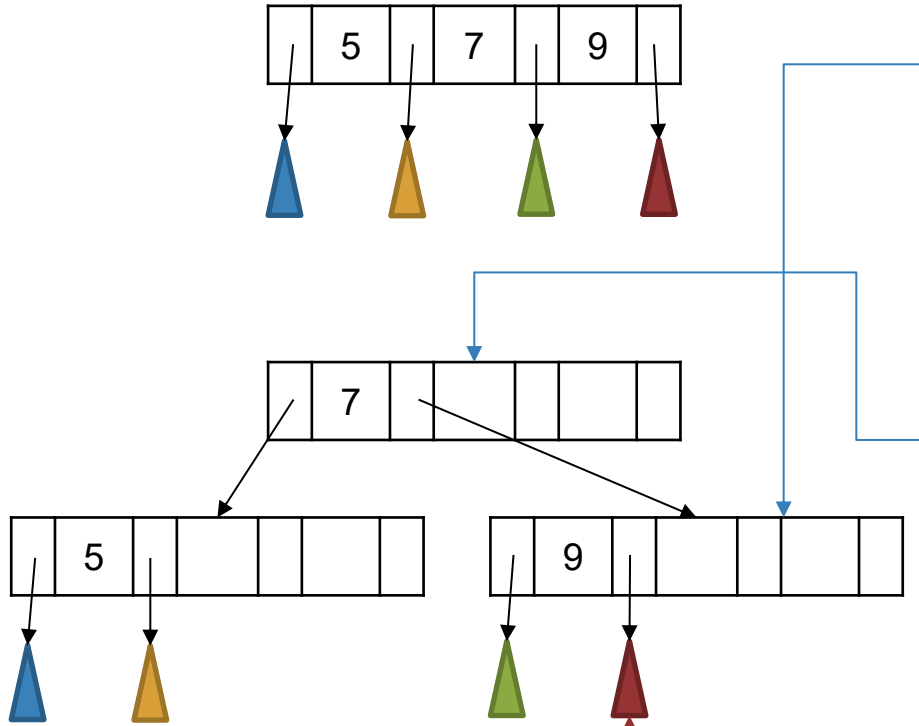



```
94   BTNode* splitChild(BTNode* root)
95   {
96       int i, mid;
97       //root 노드를 split 하게 되면
98       //      newParent
99       //  root        newSibling
100      //형태로 split이 일어나게끔 수행
101      BTNode* newParent;
102      BTNode* newSibling;
103
104      newParent = initBTNode();
105      newParent->isLeaf = 0;
106      //root 노드가 실제 root 노드였다면
107      //newParent 노드가 실제 root 노드가 된다
108      if (root->isRoot)
109          newParent->isRoot = 1;
110      root->isRoot = 0;
111
112      //newSibling 노드는 root와 같은 level 상에 존재한다
113      newSibling = initBTNode();
114      newSibling->isLeaf = root->isLeaf;
115
```



2. B-Tree – 실습

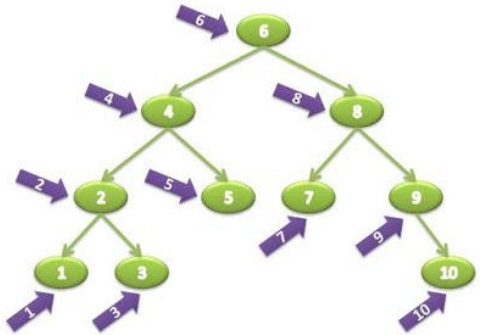
- **Split Operation** (BTNode* splitChild(BTNode* root))




```
116     //root 노드의 중간 지점을 정하고
117     //오른쪽 key 값들은 newSibling 노드로 이동
118     mid = (M_WAY - 1) / 2;
119     for (i = mid + 1; i < M_WAY; i++)
120     {
121
122
123
124
125         root->childs[i] = NULL;
126         root->keys[i] = 0;
127         root->n--;
128     }
129     //이 때 child node pointer는 키 값보다 1개 많으므로
130     //for문이 끝난 후 한 번 더 수행
131
132     root->childs[i] = NULL;
133
134     //root 노드에 중간에 위치했던 key 값을 newParent 노드로 이동
135
136
137     root->keys[mid] = 0;
138     root->n--;
139     //newParent 노드의 child 노드로 root, newSibling 노드를 연결
140
141
142
143     return newParent;
144 }
```

2. B-Tree – 실습

- **inorder traversal** (`void inorderTraversal(BTNode * root)`)

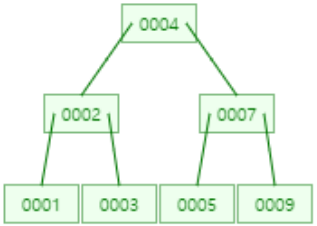



```
146  void inorderTraversal(BTNode * root)
147  {
148
149      int i;
150      printf("₩n");
151      for (i = 0; i < root->n; i++)
152      {
153          //leaf 노드가 아니라면 밑으로 탐색.
154          if (!(root->isLeaf))
155          {
156              inorderTraversal(root->childs[i]);
157              printf("  ");
158          }
159          // 데이터를 출력.
160          printf("%d", root->keys[i]);
161      }
162      //key 값보다 child 노드가 한 개 더 많으므로
163      //마지막 child 노드에 대해 밑으로 탐색
164      if (!(root->isLeaf))
165      {
166          inorderTraversal(root->childs[i]);
167      }
168      printf("₩n");
169  }
170
```

2. B-Tree – 실습

- **main**

```
C:₩WINDOWS₩system32₩cmd.exe
넣을 데이터의 개수: 7
데이터를 입력하세요: 7
데이터를 입력하세요: 5
데이터를 입력하세요: 9
데이터를 입력하세요: 3
데이터를 입력하세요: 4
데이터를 입력하세요: 2
데이터를 입력하세요: 1
트리 출력.



1
  2
3

  4

5
  7
9


계속하려면 아무 키나 누르십시오 . . .
```




```
171     int main()
172     {
173         BTNode* root;
174         int i, n, t;
175
176         root = initBTNode();
177         root->isRoot = 1;
178
179         printf("넣을 데이터의 개수: ");
180         scanf("%d", &n);
181         for (i = 0; i < n; i++)
182         {
183             printf("데이터를 입력하세요: ");
184             scanf("%d", &t);
185             root = BTInsert(root, t);
186         }
187         printf("트리 출력.₩n");
188         inorderTraversal(root);
189
190
191         return 0;
192
193     }
```

2. B-Tree – 실습

- **main**

```
선택 C:₩WINDOWS₩system32₩cmd.exe
넣을 데이터의 개수: 15
데이터를 입력하세요: 1
데이터를 입력하세요: 2
데이터를 입력하세요: 3
데이터를 입력하세요: 4
데이터를 입력하세요: 5
데이터를 입력하세요: 6
데이터를 입력하세요: 7
데이터를 입력하세요: 8
데이터를 입력하세요: 9
데이터를 입력하세요: 10
데이터를 입력하세요: 11
데이터를 입력하세요: 12
데이터를 입력하세요: 13
데이터를 입력하세요: 14
데이터를 입력하세요: 15
트리 출력.
```

```
1
  2
3
    4
5
  6
7
      8
9
  10
11
    12
13
    14
15

계속하려면 아무 키나 누르십시오 . . .
```