B-tree

**I. Introduction:**

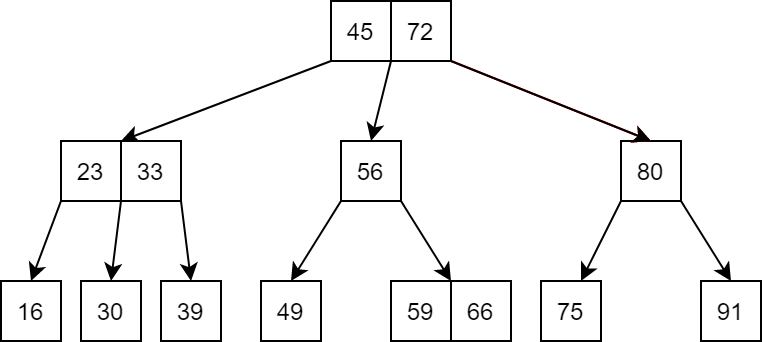
We know that implementing a balanced binary search tree is complex, and an m-way tree can still be unbalanced. To solve the problem, we will discuss about a balanced m-way tree.

B-tree is a self-balancing m-way tree. Let WAY be the maximum number of branches in a node. To keep itself balanced, a B-tree needs to satisfy:

- The root node must have at least one key.

- All nodes except root node must contain at least [(WAY – 1)/2] key, or in other word, must have at least [(WAY – 1)/2] + 1 child nodes.

- All leaves nodes are at the same level, or we can say all NULL node are at the same level.

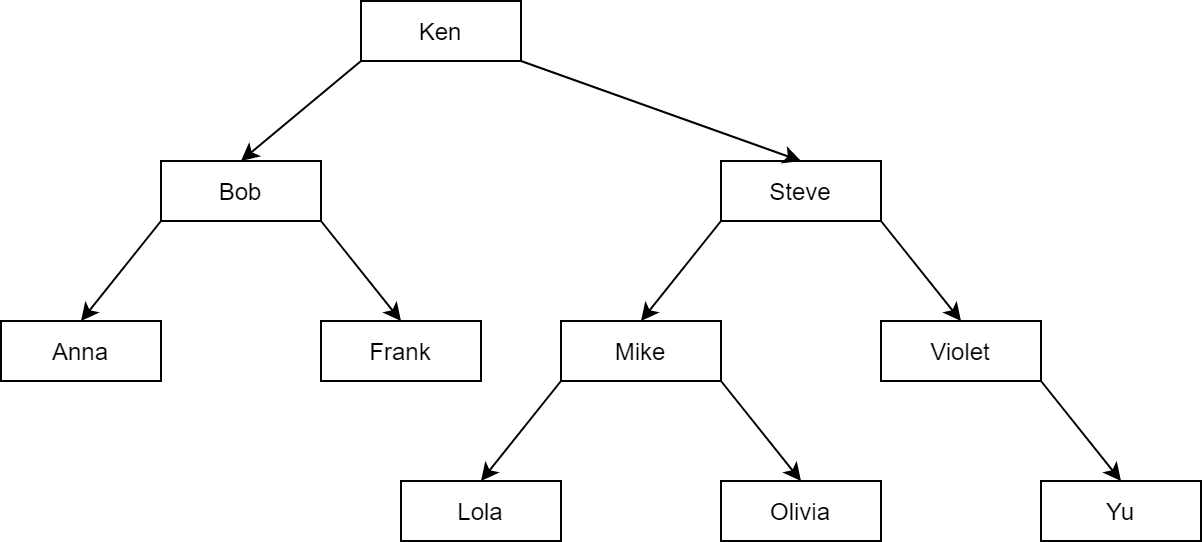
**\*Example:** A 3-way B tree with integer values :

This tree sastifies:

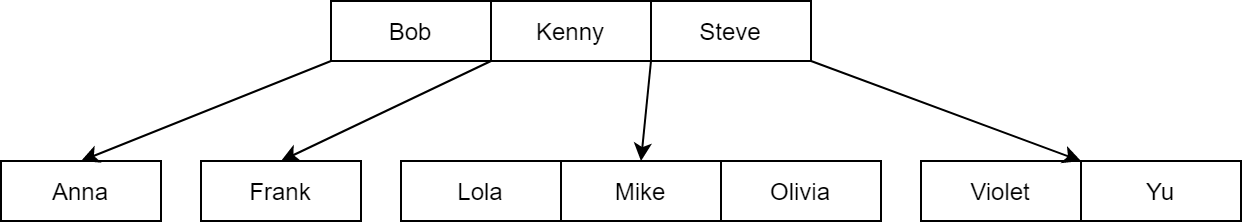
- Root node has at least 1 key. 🗸

- All nodes except root node must have at least [(3 - 1)/2] = 1 key; [(3 – 1)/2] + 1 = 2 childs. 🗸

- All leaves nodes are at the same level. 🗸

A list of name organized as a balanced BST :

With only 10 values, the BST has grown to the height of 4. When with the same amount of values, a 4-way B-tree reduce the height to 2.

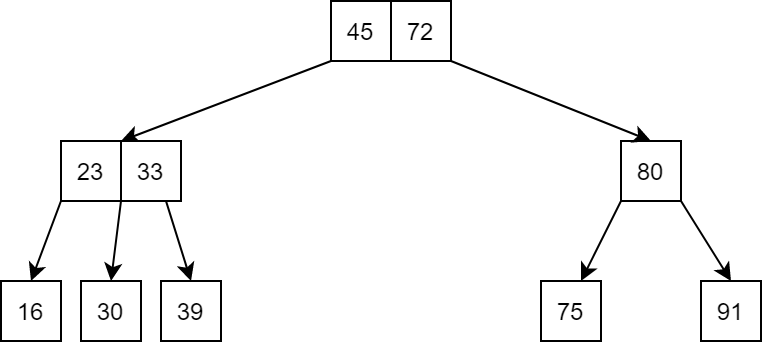


This tree sastifies:

- Root node has at least 1 key. 🗸

- All nodes except root node must have at least [(4 - 1)/2] = 1 key; [(4 – 1)/2] + 1 = 2 childs. 🗸

- All leaves nodes are at the same level. 🗸

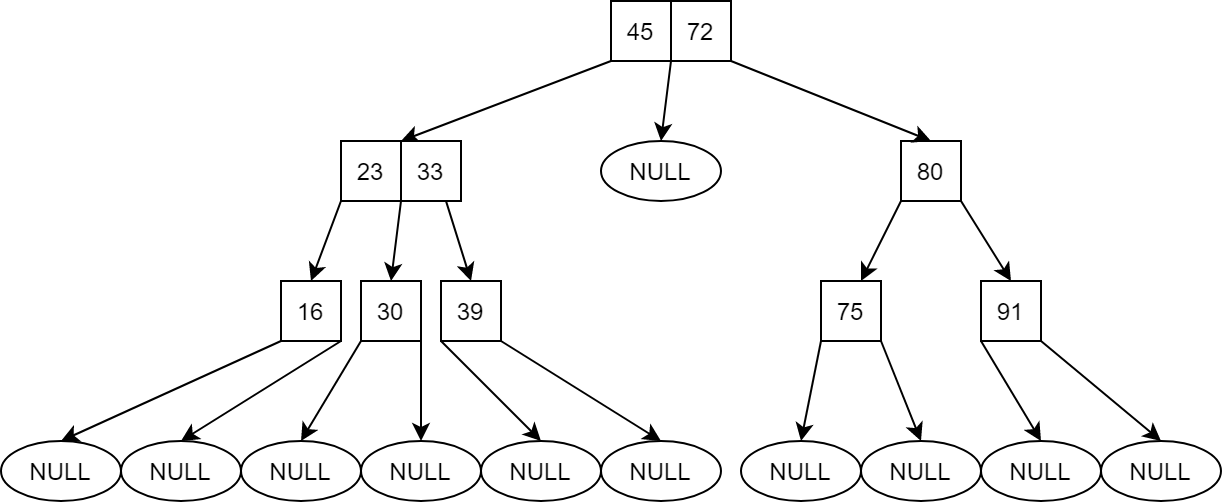
A 3-way B-tree ?

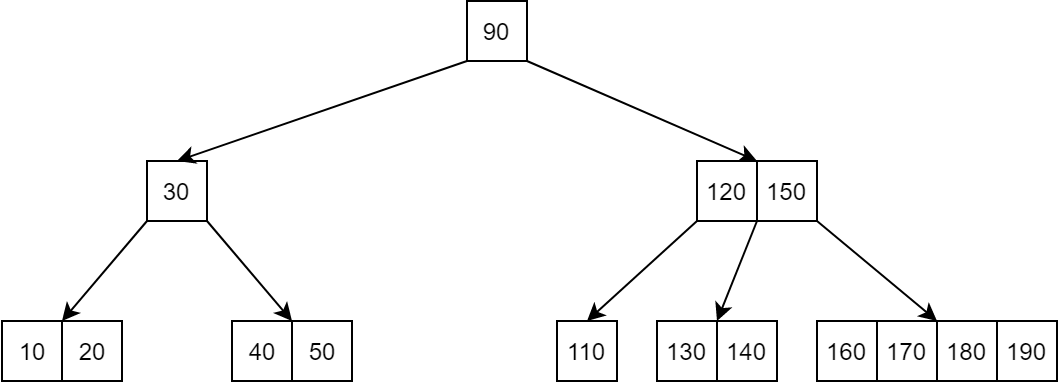
This tree sastifies:

- Root node has at least 1 key. 🗸

- All nodes except root node must have at least [(3 - 1)/2] = 1 key; [(3 – 1)/2] + 1 = 2 childs. 🗸

- All leaves nodes are at the same level. ❌ .The NULL nodes are not at the same level.



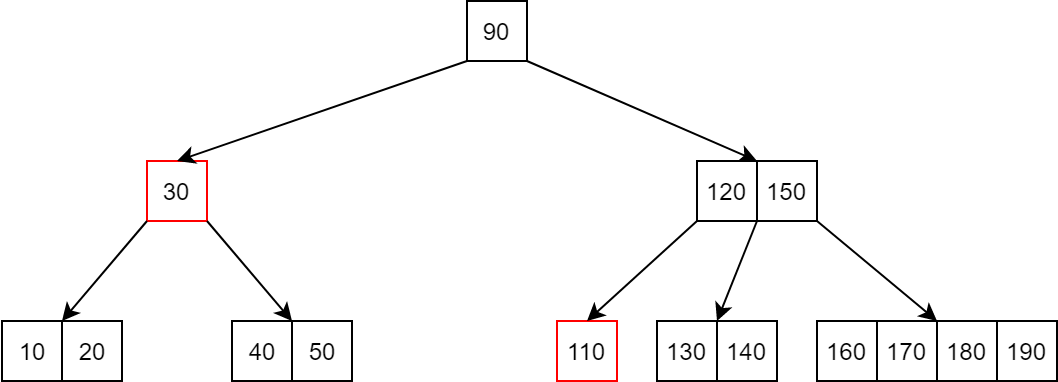
A 5-way B-tree ?

This tree sastifies:

- Root node has at least 1 key. 🗸

- All leaves nodes are at the same level. 🗸

- All nodes except root node must have at least [(5 - 1)/2] = 2 key; [(5 – 1)/2] + 1 = 3 childs. ❌ .There are nodes with not enough keys and childs.



A B-tree node can be define like an m-tree node, but let’s add a boolean value to determine if a node is leaf node or not, and a parent node address for easy traversing. Note that this is not the best way to define the structure of a B-tree node, it is only used to explain clearly the idea of the algorithms later. In this structure, we add one extra slot to the *value* and *child* array, which we will used later.

#define WAY 3 // maximum number of branches in a node

#define MIN ((WAY - 1)/2) // minimum number of keys in a node

struct node

{

int count;

int value[WAY];

struct node\* child[WAY + 1];

struct node\* parent;

bool isLeaf;

};

**II. Algorithms:**

**1. Traversing and searching algorithms:**

Because B-tree is an m-way tree, the traversing and searching method is exactly the same as m-way tree. So, we bring back the *search\_in\_node* function introducted in the m-way tree section. This function return a boolean value and an integer value *pos,* they are either 1 and position of key if found in current node, or 0 and position of the child node that might contain key.

bool search\_in\_node(int key, node\* curr, int& pos)

{

pos = 0;

// find the first key in node that is >= key,

while ((pos < curr->count) && (key > curr->value[pos])) pos++;

// if key > all values in node return 0, pos is position of the last child in node

if (pos == curr->count) return 0;

// if key is found, return 1, pos is now position of key in current node;

if (key == curr->value[pos]) return 1;

// else return 0, pos is now position of the child node that might contain key

return 0;

}

node\* search(int key, node\* curr, int& pos)

{

// reach NULL node mean the key is not found in the tree, return NULL

if (curr == NULL) return NULL;

// if key is found in the current node return the current node

// pos will be position of key in current node

if (search\_in\_node(key, curr, pos)) return curr;

// child[pos] is now the possible child to contain key, call recursive to this child

return search(key, curr->child[pos],pos);

}

**2. Inserting algorithms:**

A new key always added to a B-tree at a leaf node.

To add a new key to the B-tree we follow these steps:

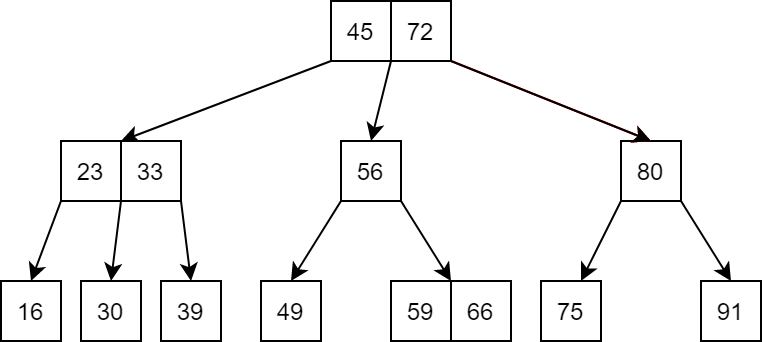
- Traverse the tree to reach a suitable leaf node.

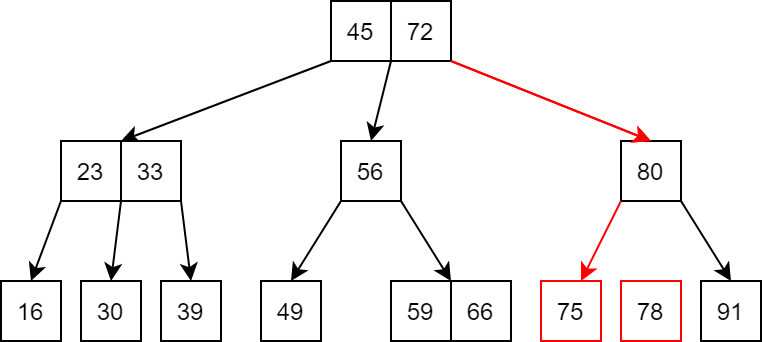
- Add new key to the leaf node.

- If the leaf node is full, splits it in half and moves the middle value onto its parent node.

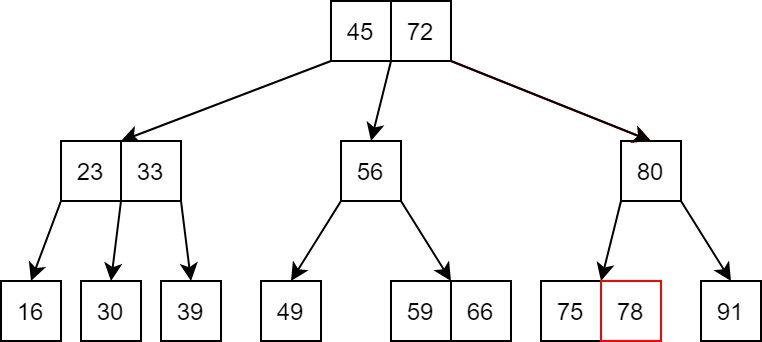
- Call recursive to check the parent node and split it if it is full until we reach a non-full node or root node.

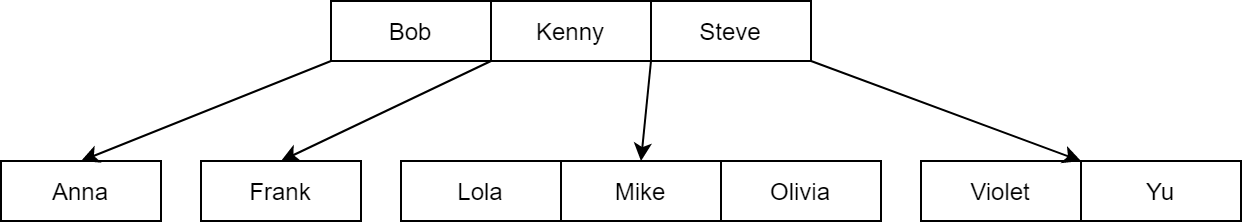
- If the root node is full, splits it in half and creates a new root that will contain the middle value. This is the reason B-tree grow upward, unlike the BST which grow downward.

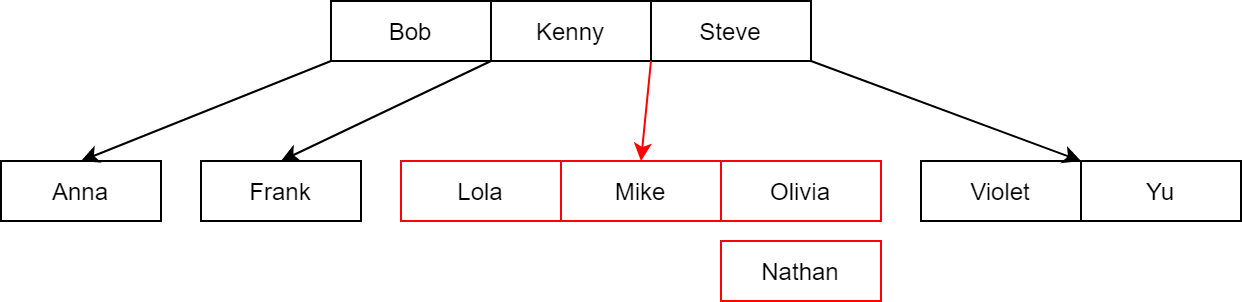
**\* Example:** Add 78 to this 3-way B-tree:

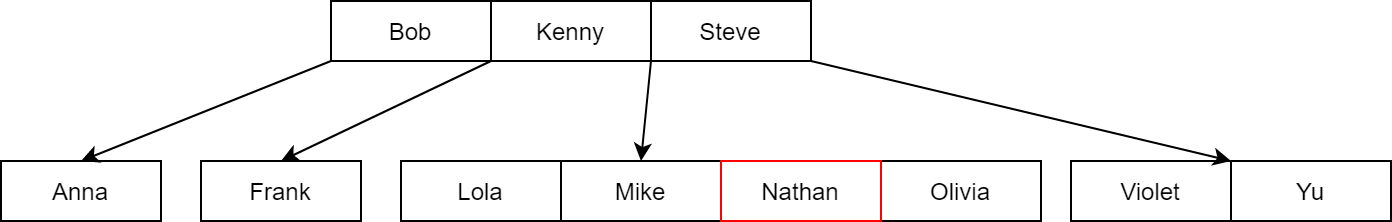
Start from the root node, we traverse the tree to reach a leaf node that will have 78 insert to.

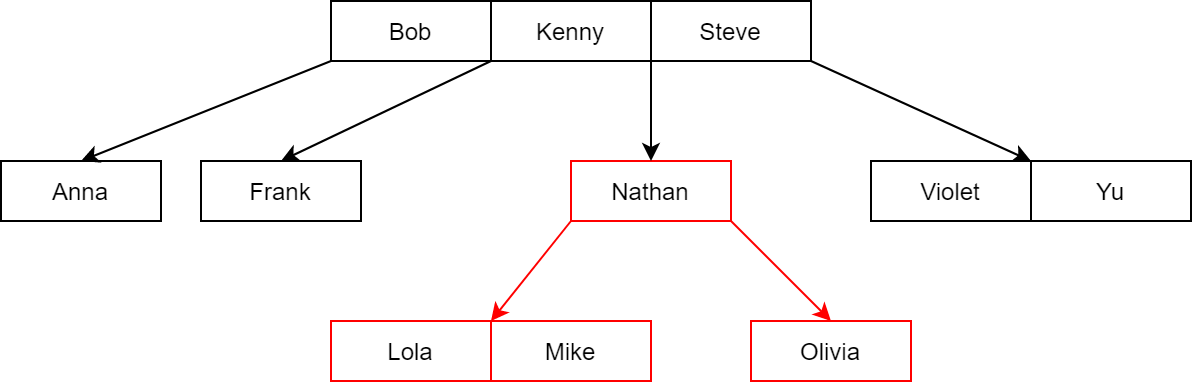
75 < 78, so we add 78 after 75.The leaf node is not full, no adjustment needed.

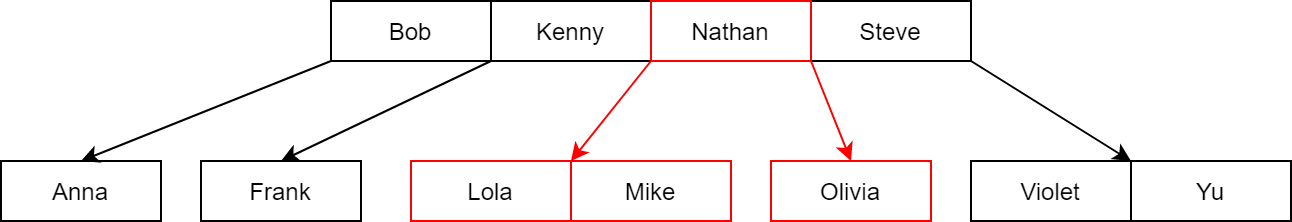


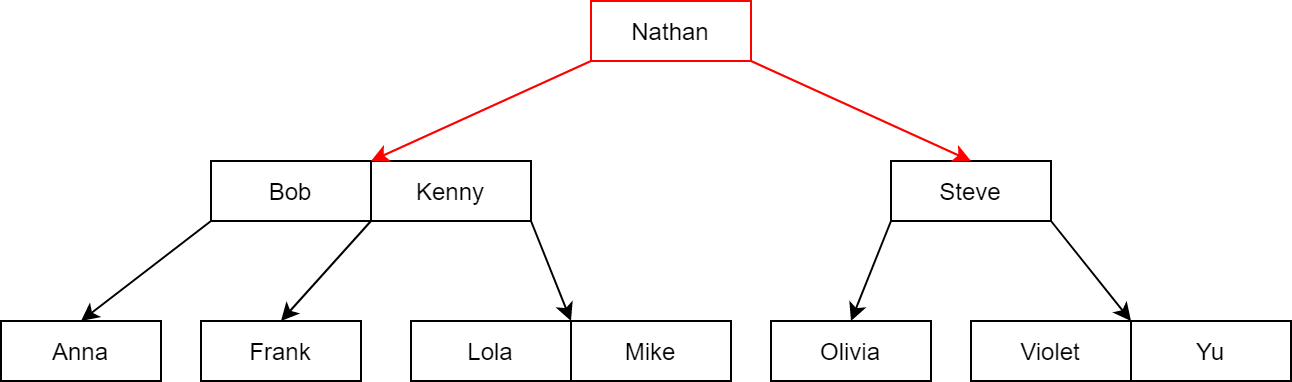
Now let’s add a new name, Nathan, to this 4-way B-tree.

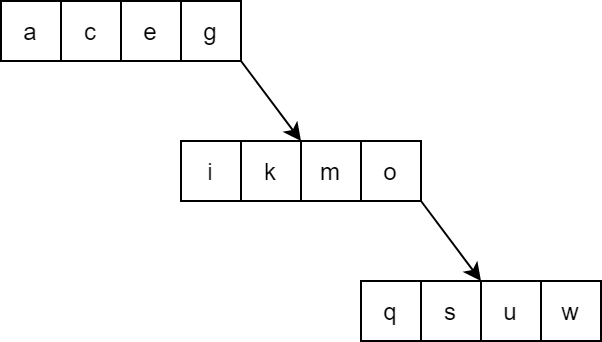


Mike < Nathan < Olivia, so Nathan is added to the middle.

The leaf node is full, we split it in half and move the middle value, Nathan to the parent node, or in other word, insert Nathan to the parent node. Note that in this case, the middle value can be either Mike or Nathan, depend on how we design the algorithms.

Kenny < Nathan < Steve

Now the root node is also full, we split it in half and the middle value,Nathan , become the new root. We can see the tree grow upward.

In the m-way tree section, we introduced an unbalanced 5-way tree with characters values :

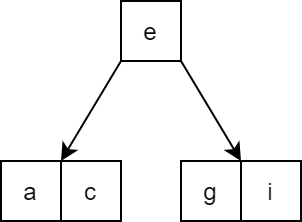
Now let’s make a 5-way B-tree from these values.

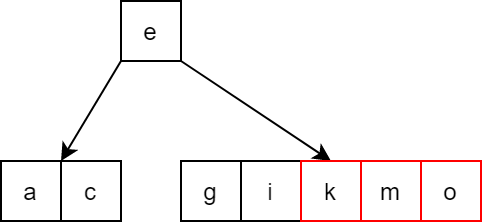
At first we have an empty tree, the first value will be added to the new-created root node, which is also the leaf node.

Continue on adding c, e and g. The root node will contain 4 keys.

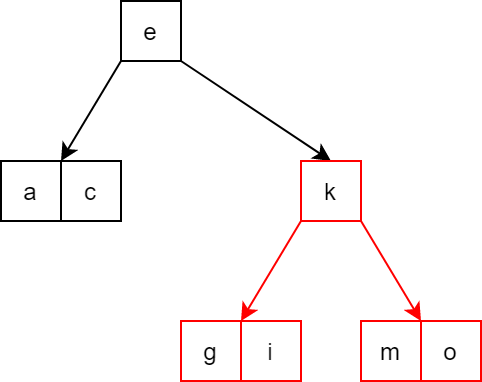
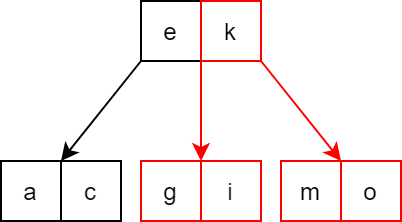
Now as we try to add i, we first add it to the root node.

The root node is now full, we split it in half and e become the new root.

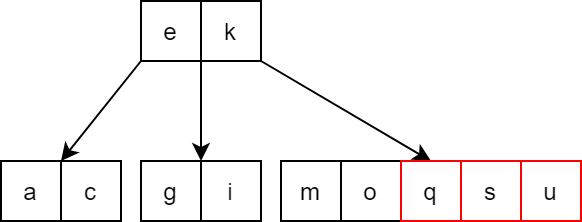
A node in 5-way B-tree other than root node must have at least [(5 – 1)/2] = 2 keys.

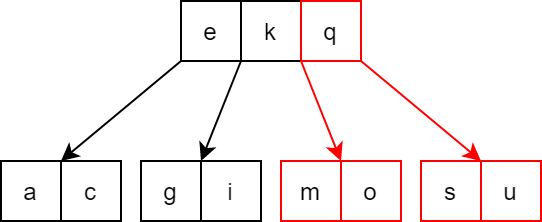
Continue on adding k and m, then o.

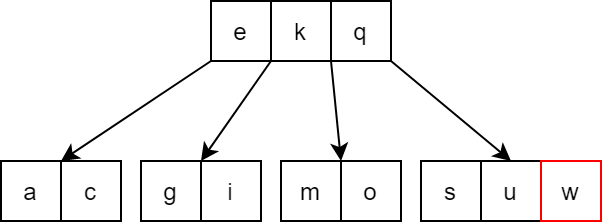
The leaf node is now full, we split it in half and k move to the parent node.



The same thing happen as q,s and u is inserted.



Split and move q to parent:

Now we finish by adding w.

If we add values to a normal m-way tree in sorted order, the tree keep growing in height and become totally unbalanced, while B-tree rebalance itself.

The inserting algorithms is implement in C++ below. This is NOT the best way to do it, even bad, but it follow the idea of the algorithms step by step for easy understanding.

int findChildPos(node\* child, node\* parent)

{

// find the position of child in parent's child array

for (int i = 0; i <= parent->count; i++)

{

if (parent->child[i] == child) return i;

}

return -1;

}

void insert\_at\_leaf(node \*curr,int key)

{

// add the new key to the leaf node

int i = 0;

// find position for key

while ((curr->value[i] < key)&&(i < curr->count)) i++;

// shift value > key to the right

for(int j = curr->count; j > i;j--)

{

curr->value[j] = curr->value[j - 1];

}

curr->value[i] = key; // add key

curr->count++; // update count value

}

void split\_node(node\* parent, int childpos)

{

// split child in half and move middle element to parent

node\* c1 = parent->child[childpos];

// create new node for the second half

node\* c2 = new node;

c2->isLeaf = c1->isLeaf;

// move MIN elements to the second node

c2->count = MIN;

for (int i = 0; i < MIN; i++)

{

c2->value[i] = c1->value[WAY - MIN + i];

c2->child[i] = c1->child[WAY - MIN + i];

if (c2->child[i] != NULL) c2->child[i]->parent = c2;

}

c2->child[c2->count] = c1->child[c1->count];

if (c2->child[c2->count] != NULL) c2->child[c2->count]->parent = c2;

int midvalue = c1->value[WAY - MIN - 1];

// if the child is not root node

if (parent != NULL)

{

// move middle element to parent

for (int i = parent->count; i > childpos; i--)

{

parent->value[i] = parent->value[i - 1];

parent->child[i + 1] = parent->child[i];

}

parent->value[childpos] = midvalue;

parent->child[childpos + 1] = c2;

parent->count++;

c2->parent = parent;

}

c1->count -= (MIN + 1);

}

void insert\_key(node\*& root, int key)

{

// if tree is empty create a new root and add key to it

if (root == NULL)

{

root = create\_node();

root->isLeaf = 1;

root->count = 1;

root->value[0] = key;

return;

}

node\* temp = root;

// find a leaf node to insert key

while (!temp->isLeaf)

{

int pos;

if (search\_in\_node(key, temp, pos)) return;

else temp = temp->child[pos];

}

// add key to found leaf node

insert\_at\_leaf(temp, key);

for (int i = 0; i <= WAY;i++) temp->child[i] = NULL;

// check if the node is full and split if full

// repeat which the parent node until the node is not full or reached root node

while (temp->parent != NULL)

{

if (temp->count == WAY)

{

split\_node(temp->parent, findChildPos(temp, temp->parent));

temp = temp->parent;

}

else break;

}

// if root node is full

if (temp->count == WAY)

{

// repeat the first half of splitChild function

node\* c1 = temp;

node\* c2 = new node;

c2->isLeaf = c1->isLeaf;

c2->count = MIN;

for (int i = 0; i < MIN; i++)

{

c2->value[i] = c1->value[WAY - MIN + i];

c2->child[i] = c1->child[WAY - MIN + i];

if(c2->child[i] != NULL) c2->child[i]->parent = c2;

}

c2->child[c2->count] = c1->child[c1->count];

if (c2->child[c2->count] != NULL)

c2->child[c2->count]->parent = c2;

int midvalue = c1->value[WAY - MIN - 1];

// create new root and move middle element to it

node\* newroot = create\_node();

newroot->count = 1;

newroot->value[0] = midvalue;

newroot->child[0] = c1;

newroot->child[1] = c2;

c1->parent = newroot;

c2->parent = newroot;

root = newroot;

c1->count -= (MIN + 1);

}

return;

}

**3. Deleting algorithms:**

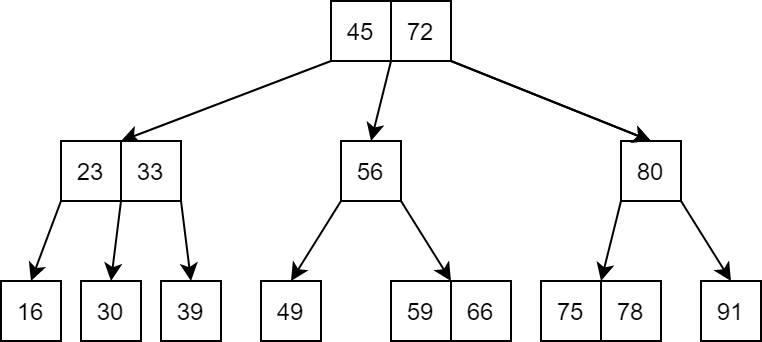
To delete a key from a B-tree, first we do the same as a normal m-way tree. The actual node to have a key remove from will always be the leaf node.

If a node has less than [(m – 1)/2] keys we do one of the following:

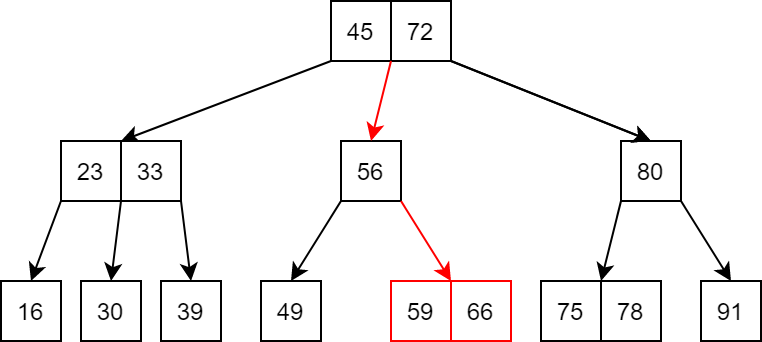
- If an adjacent sibling node has enough key, borrow 1 key from it to replace a key in parent node and add that key to the current node.

- Otherwise, merge the current node with an adjacent sibling node and a corresponding key from parent node.

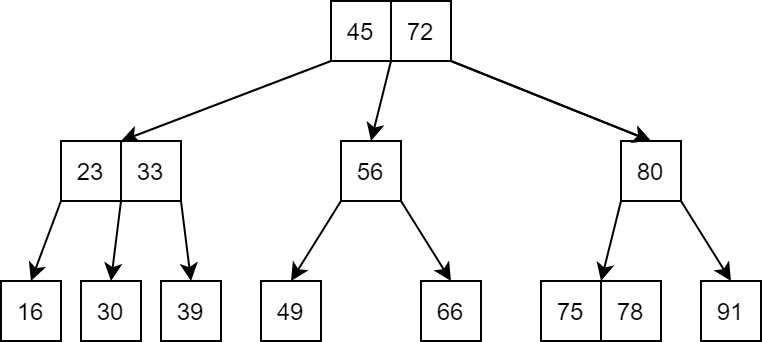
**\* Example:**

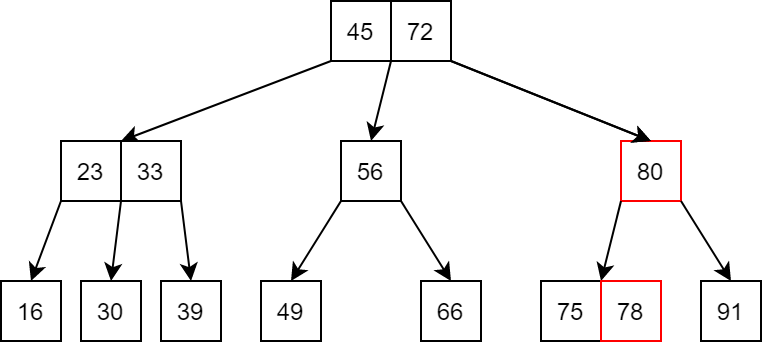


Remove 59 and 80 from the above 3-way B-tree.

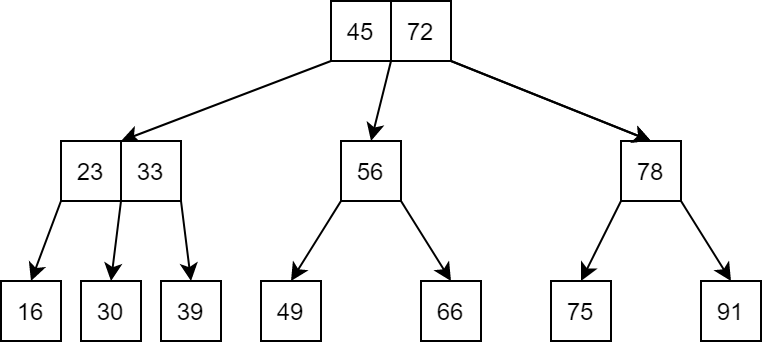
We traverse the tree to find the node contain 59. 59 doesn’t have any child, so we just remove it.

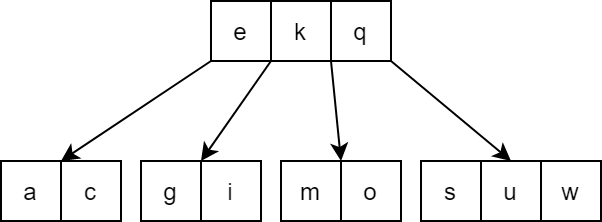
After removing 59, the node still have (3-1)/2 = 1 key, no adjustment needed.

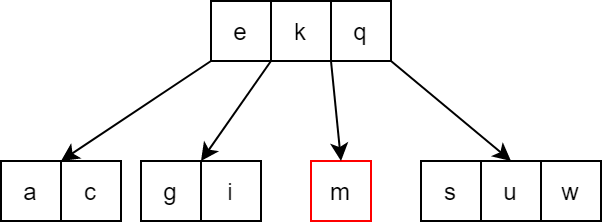


Now we find the node contain 80. 80 has both left and right child, we can choose to replace 80 with the largest value in the left sub tree, or the smallest value in the right sub-tree. We choose the left one this time. The largest value in the left sub tree is 78.

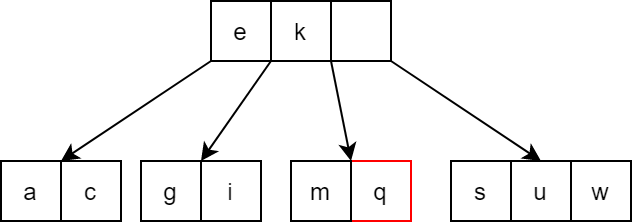
After removing 78 from its old node, the node still have enough key, no adjustment needed.

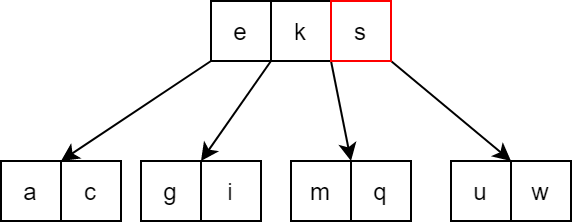


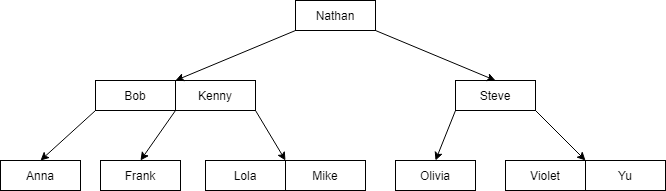
Now let’s try removing o from this 5-way B-tree:

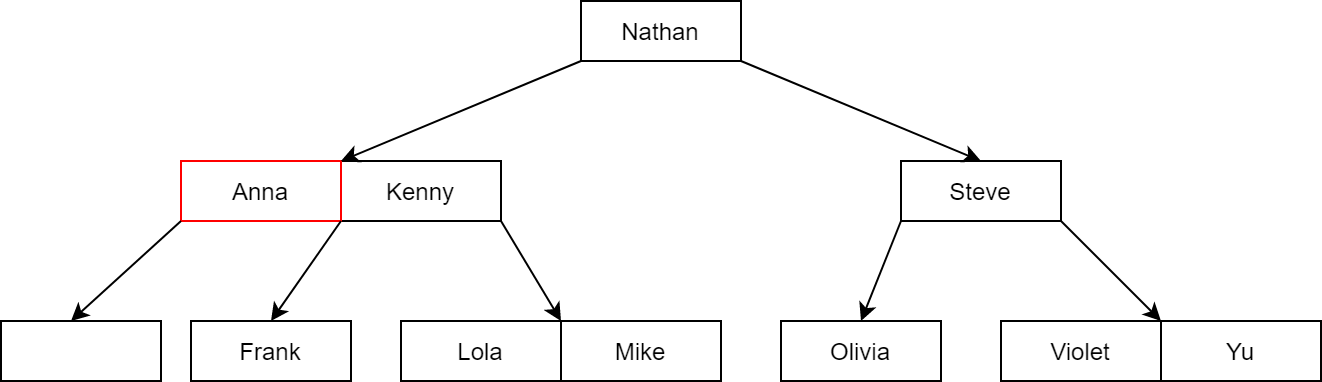
We find the node holding o, but after we remove o from it, it only have 1 key left, which is less than (5 – 1)/2 = 2 keys.

The left adjacent sibling node doesn’t have enough key to share, but the right one does, so we borrow s from the right sibling node. We start by moving the key in parent node that is in between current node and right sibling node down to current node.



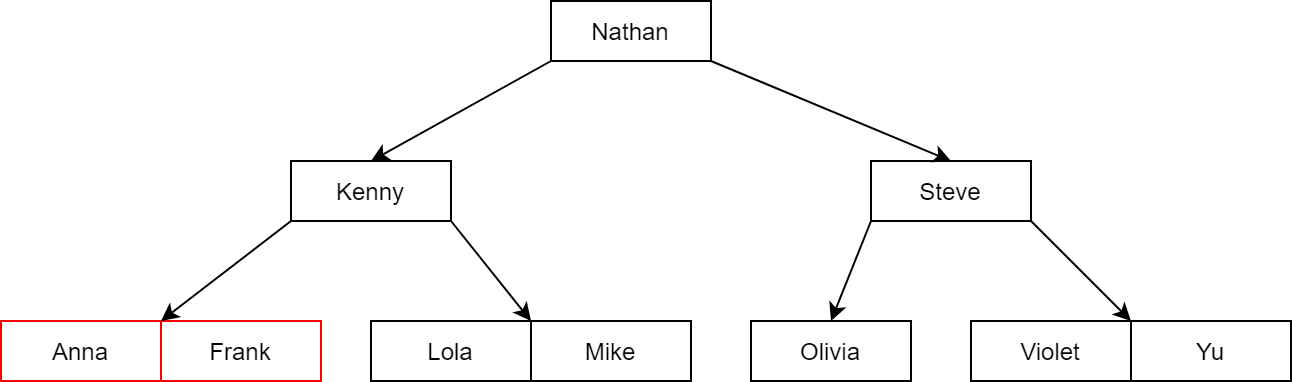
Now the borrowed value in the right sibling node move up to free slot in the parent node.

Now let’s remove Bob from this 4-way B-tree.

Bob has both child, we can replace him with the largest value in the left sub tree, Anna.

Now Anna’s old node doesn’t have enough key. Because the node is the first child of its parent node, it only have right adjacent sibling node. But the right sibling node doesn’t have enough key to share, so we much do the merging operation.

Merge the current node, the right sibling, and the key in between them in parent node into one node and place it in the old node position.



For deeper understanding, we separate the fixing operation after deleting into 4 case. In the following part:

- P represent keys of parent node.

- S represent keys of siblling node.

- N represent keys of current node.

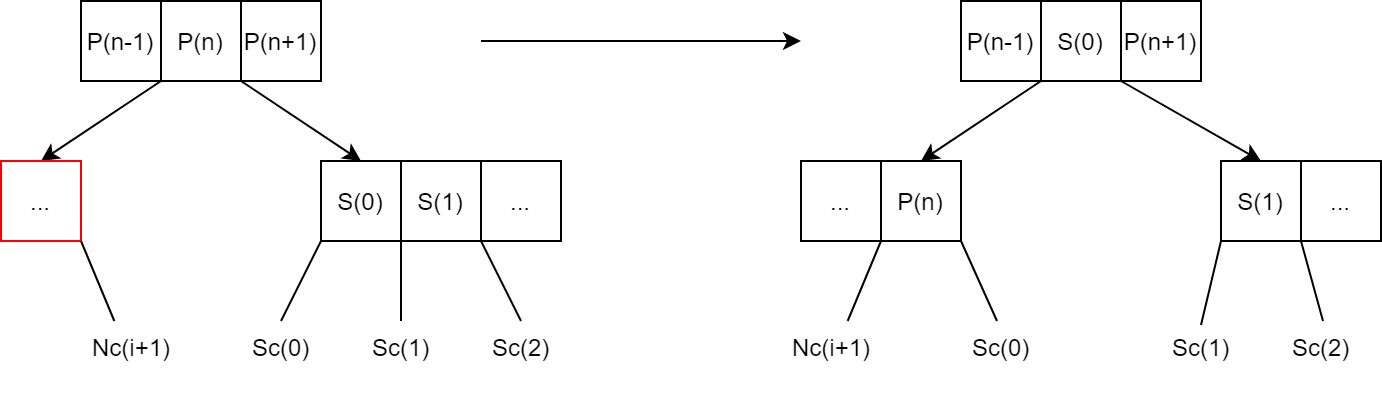
- Sc represent children of sibling node.

- Nc represent children of current node.

- (…) represent 0 or more keys and corresponding child.

- The red node is the current node that missing keys, it can be empty or not, but always have at least one child.

Case 1: Borrow from right sibling

Pseudo code:

void fixCase1(node \*N, node \*S, node \*P,int n)

{ N[N->count] = P[n] ;

P[n] = S[0];

N->child[N->count + 1] = S->child[0];

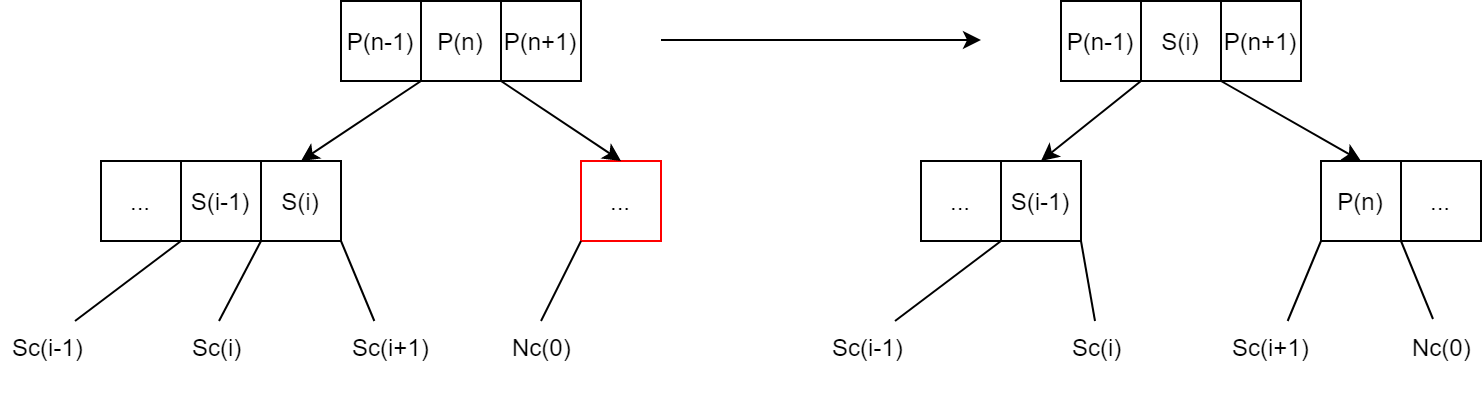
for i = 0 to S->count – 2 : S[i] = S[i+1]; S->child[i] = S->child[i+1];

S->child[S->count - 1] = S->child[S->count];

S->count = S->count – 1;

N-> count = N-> count + 1; }

Case 2: Borrow from left sibling

Pseudo code:

void fixCase2(node \*N, node \*S, node \*P,int n)

{ for i = N->count downto 1 : N[i] = N[i-1]; N->child[i+1] = N->child[i];

N->child[1] = N->child[0];

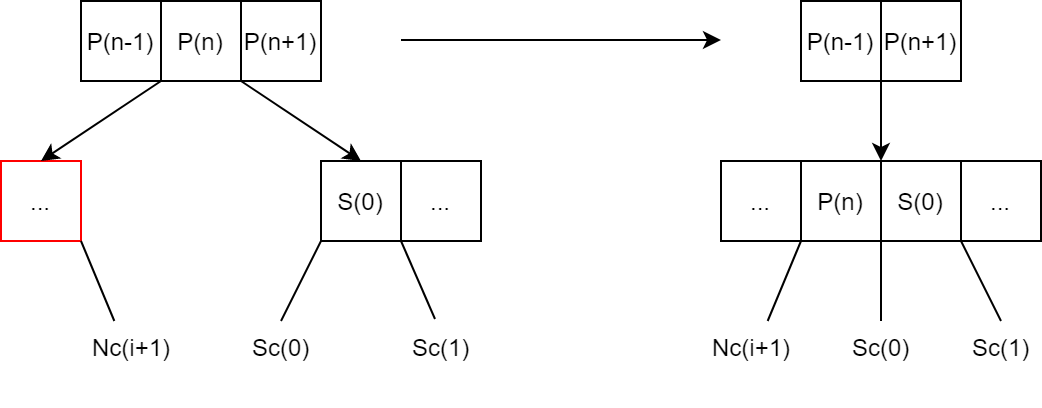
N[0] = P[n];

P[n] = S[S->count – 1];

N->child[0] = S->child[S->count];

S->count = S->count – 1;

N-> count = N-> count + 1; }

Case 3: Merge with right sibling

Pseudo code:

void fixCase3(node \*N, node \*S, node \*P,int n)

{ N[N->count] = P[n];

N->count = N->count + 1;

for i = 0 to S->count – 1 :

{ N[N->count] = S[i];

N->child[N->count] = S->child[i];

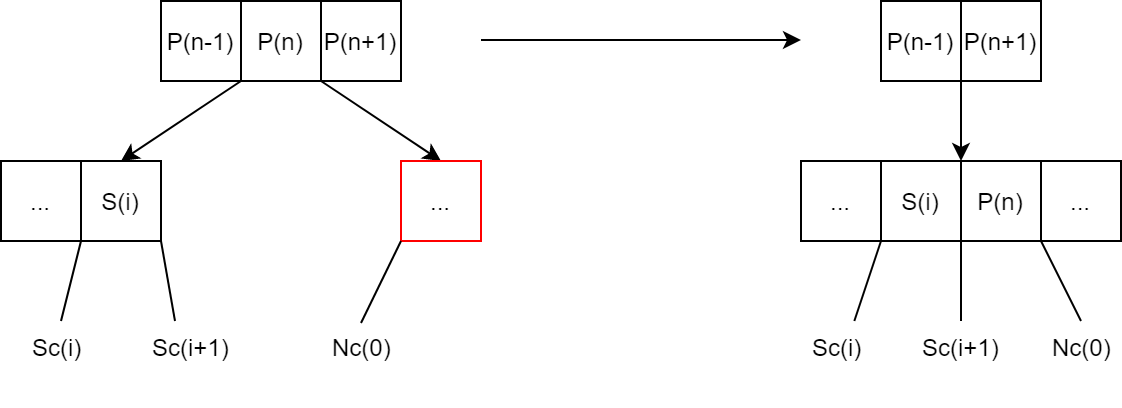
N->count = N->count + 1; }

N->child[N->count] = S->child[S->count];

for i = n to P->count – 2 : P[i] = P[i+1]; P->child[i+1] = P->child[i+2];

P->count = P->count – 1;

delete S; }

Case 4: Megre with left sibling

Pseudo code:

void fixCase4(node \*N, node \*S, node \*P,int n)

{ S[S->count] = P[n];

S->count = S->count + 1;

for i = 0 to N->count – 1 :

{ S[S->count] = N[i];

S->child[S->count] = N->child[i];

S->count = S->count + 1; }

S->child[S->count] = N->child[N->count];

for i = n to P->count – 2 : P[i] = P[i+1]; P->child[i+1] = P->child[i+2];

P->count = P->count – 1;

delete N; }