1 B$^+$ tree

Motivation for B$^+$ tree is to have data structure with the properties as for B tree, while keys can be accessed in batches. Thus, for each key the adjacent keys can be found in constant time.

1.1 Definition

B$^+$ tree is B tree with the additional requirements:

1. All keys are stored in the leaves.
2. Leaves form a linked list starting from the leftmost leaf. It is called *sequence set*.
3. Internal nodes do not necessarily keep all of the keys. Those that are present, separate keys of the children in the same way as in B tree. They form so called *index*.

So, while B tree stores keys in all nodes (internal and leaves), B$^+$ tree keeps all keys in leaves and some of them in internal nodes. In addition, all leaves are linked into one single linked list.

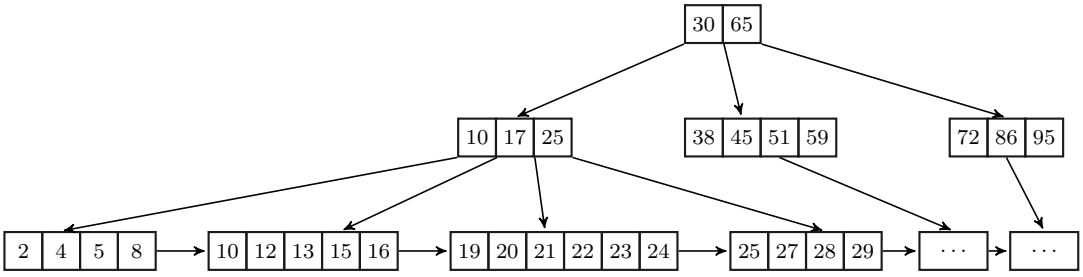

Figure 1: Example of B$^+$ tree of degree $t = 4$

The figure 1 shows an example of B$^+$ tree where 17 is not the key since it is present in an internal node only, while 10 and 25 are keys which also occur in internal nodes.

In the pseudo code, all notations remain same as for B tree. Additionally, each leaf x has a pointer $\mathsf{next}(x)$ to a next leaf.

1.2 Searching

Starting from the root of a B$^+$ tree, the algorithm finds appropriate child as in the case of B tree. If key K is found in an internal node, then the search is not stopped, but the appropriate right pointer is chosen, so the algorithm proceeds down to a leaf.

Input: key K to find in subtree x with n elements
Output: node which contains K or null
Complexity: $O(\log n)$

```
find(K, x)
  z = x
  while z ≠ null
    i = 1
    while i ≤ keys_no(z) and K > key(i, z)
      i = i + 1
    if i ≤ keys_no(z) and K = key(i, z) and leaf(z)
      break {z found}
```

```
    if leaf(z)
      z = null {z not found}
    else
      z = read(child(i, z)) {get child from external memory}
  return z
```

From the algorithm follows that it doesn't matter which keys are stored in internal nodes, as long they separate keys in leaves in a proper way.

1.3 Auxiliary node operations

Splitting node is performed in a similar manner as in B tree. The difference is that central key is copied to a parent node and placed in the right sibling. Additionally, the sequence set is updated if necessary.

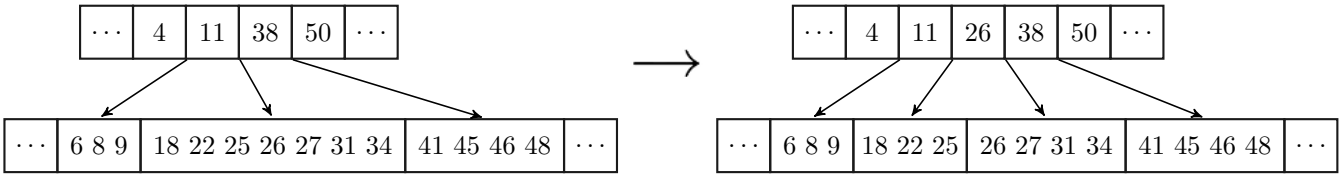

Figure 2: splitting 7-elements node ($t = 4$), key 26 is copied to the parent

The split method is slightly modified to support copying of the central key to both parent and sibling node.

Input: node x of degree t, with a full child at i-th position
Output: none
Complexity: $O(t)$

```
split(x, i)
  y = child(i, x) {full node}
  new z
  leaf(z) = leaf(y)
  keys_no(z) = degree(T) − 1
  {copy second half of keys from y to z, including central key}
  for j = 1 to degree(T)
    key(j, z) = key(degree(T) − 1 + j, y)
  {copy second half of children from y to z, including central key}
  if not leaf(y)
    for j = 1 to degree(T) + 1
      child(j, z) = child(degree(T) − 1 + j, y)
  keys_no(y) = degree(T) − 1
  {move x's children one place to the right to make room for z}
  for j = keys_no(x) + 1 downto i + 1
    child(j + 1, x) = child(j, x)
  child(i + 1, x) = z
  {add new key key(t, y) for z into x}
  for j = keys_no(x) downto i
    key(j + 1, x) = key(j, x)
  key(i, x) = key(t, y)
  keys_no(x) = keys_no(x) + 1
  {update sequence set if necessary}
```

if leaf(y)
  next(z) = next(y)
  next(y) = next(z)
write(x)
write(y)
write(z)

Merging is similar to the one on B tree except that central key is not copied from parent to the merged children. Additionally, the sequence set is updated if necessary.

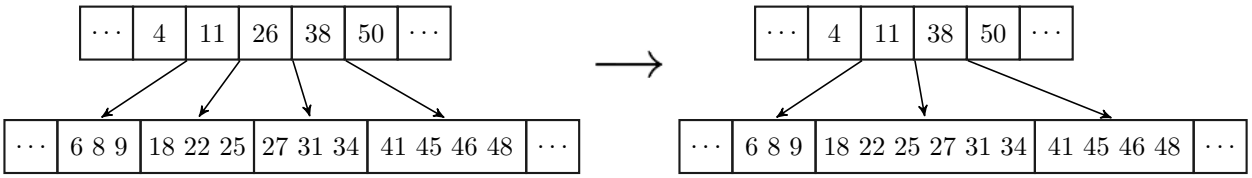

Figure 3: merging two 3-elements nodes ($t = 4$)

Input: index i of the node x (with degree t) to merge children c_i and c_{i+1}
Output: none
Complexity: $O(t)$
merge(x, i)
  y = child(i, x), z = child($i + 1, x$)
  {*move i-th key of x into y*}
  key(t, y) = key(i, x)
  {*move the rest of x's keys to the left*}
  for $j = i$ **to** keys_no(x)
    key(j, x) = key($j + 1, x$)
  delete key($n(x) + 1, x$)
  keys_no(x) = keys_no(x) $- 1$
  {*copy z's keys into y*}
  for $j = 1$ **to** degree(T) $- 1$
    key($t + j, y$) = key(j, z)
  {*copy z's children into y*}
  if not leaf(z)
    for $j = 1$ **to** t
      child($t + j, y$) = child(j, z)
  $n(y) = 2 \cdot$ degree(T) $- 1$
  {*update sequence set if necessary*}
  if leaf(y)
    next(y) = next(z)
  delete z
  {*remove link for z from x*}
  delete child($i + 1, x$)
  for $j = i + 1$ **to** keys_no(x)
    child(j, x) = child($j + 1, x$)
  delete child(keys_no(x) $+ 1, x$)
  write(x)
  write(y)
  write(z)

Moving key $K \in c_i$ is performed in a manner similar to the B tree's move. K replaces the correspond-

ing parent key which splits c_i and c_{i+1} and K is copied to c_{i+1} to be it's first key.

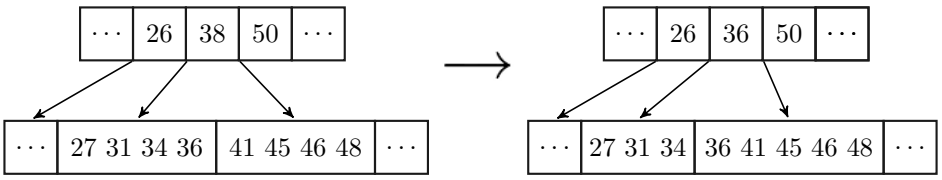

Figure 4: moving key 36

Input: node x with a key at i-th place and degree t, its children c_i and c_{i+1} with degrees at least t and at most $2t-2$, respectively
Output: key from c_i moved to parent and parent key moved to c_{i+1}
Complexity: $O(t)$

```
move_key_next(x, i)
  a = child(i, x)
  b = child(i + 1, x)
  {move keys right to make room for the moving one}
  for j = 1 to keys_no(b)
    key(j + 1, b) = key(j, b)
    if not leaf(b)
      child(j + 1, b) = child(j, b)
  keys_no(b) = keys_no(b) + 1
  key(1, b) = key(i, x) = key(keys_no(a) + 1, a)
  child(1, b) = child(keys_no(a) + 1, a)
  delete key(keys_no(a) + 1, a)
  delete child(keys_no(a) + 1, a)
  keys_no(a) = keys_no(a) − 1
  write(x)
  write(a)
  write(b)
```

Input: node x with key at i-th place and degree t, its children c_i and c_{i-1} with degrees at most $2t-2$ and at least t, respectively
Output: key from c_{i+1} moved to parent and parent key moved to c_i
Complexity: $O(t)$

```
move_key_prev(x, i)
  a = child(i, x)
  b = child(i − 1, x)
  keys_no(b) = keys_no(b) + 1
  key(keys_no(b), b) = key(i, x) = key(1, a)
  child(keys_no(b) + 1, b) = child(1, a)
  {move keys left to fill empty slot}
  for j = 2 to keys_no(a)
    key(j − 1, a) = key(j, a)
    if not leaf(a)
      child(j − 1, a) = child(j, a)
  delete key(keys_no(a) + 1, a)
  delete child(keys_no(a) + 1, a)
  keys_no(a) = keys_no(a) − 1
  write(x)
  write(a)
  write(b)
```

1.4 Insert

Inserting node is exactly the same as for B tree, except the modified split for B^+ tree is used.

1.5 Delete

Deleting key K is easier than in case of B tree, because all keys are in leaves. If K is in the index only, it is not deleted, because it keeps to separate keys in the index in a proper way. Thus, the following cases are distinguished:

D1 If the key K is in leaf x, then delete the key K from x.

D2 Find child c_i that contains K. If c_i has only $t-1$ keys, execute step D2.1 or D2.2 as necessary to guarantee that we descend to a node containing at least t keys. Then, recursively delete K on c_i.

- **D2.1** If c_i has only $t-1$ keys but has an immediate sibling with at least t keys, move key from sibling to c_i.
- **D2.2** If c_i and both of c_i's immediate siblings have $t-1$ keys, merge c_i with one sibling.

Input: key K to delete in subtree $x \in T$, where T has n nodes
Output: node from which the key K is deleted
Complexity: $O(\log n)$

```
delete(x, K)
  i = index(K, x)
  if i ≠ null
    if leaf(x) {case D1}
      for j = i to keys_no(x) + 1
        key(j, x) = key(j + 1, x)
      delete key(keys_no(x) + 1, x)
      keys_no(x) = keys_no(x) − 1
      write(x)
  else {case D2}
    i = index_child(K, x)
    if keys_no(child(i, x)) = degree(T) − 1
      if 1 < i < keys_no(x) + 1
        if keys_no(child(i − 1, x)) ≥ degree(T) {case D2.1}
          move_key_next(x, i − 1)
        else if keys_no(child(i + 1, x)) ≥ degree(T) {case D2.1}
          move_key_prev(x, i + 1)
        else {case D2.2}
          merge(x, i)
      else if i = 1
        if keys_no(child(i + 1, x)) = degree(T) − 1 {case D2.2}
          merge(x, i)
        else {case D2.1}
          mode_key_prev(x, i + 1)
      else if i = keys_no(x) + 1
        if keys_no(child(i − 1, x)) = degree(T) − 1 {case D2.2}
```

```
            merge(x, i − 1)
          else {case D2.1}
            move_key_next(x, i − 1)
        delete(child(i, x), K)
      else
        delete(child(i, x), K)
  return x
```

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms; Second Edition

[2] Miodrag Zivkovic: Algoritmi

[3] Robert Sedgewick: Algorithms

[4] Daniel Dominic Sleator, Robert Endre Tarjan: Self-Adjusting Binary Search Trees

[5] Douglas Comer: The Ubiquitous B-Tree