1 B tree

Motivation for B tree is to have data structure that seldom reads or writes keys from the external memory. When B tree does the read or write, keys are taken in batches, so the communication with the external memory is minimized. Operations of interest are finding, inserting and deleting key.

1.1 Definition

B tree T with a root r_T is a tree with the following properties:

1. Every node x has the following fields:
   (a) n – number of keys currently stored in node x.
   (b) k_i – keys stored in nondecreasing order, so that $k_1 \leq k_2 \leq \ldots \leq k_n$.
   (c) l – boolean which is true if x is a leaf and false if x is an internal node.
2. Each internal node x contains $n+1$ children $c_1, c_2, \ldots, c_{n+1}$. Leaf nodes have no children, so those fields are null.
3. The keys k_i separate the ranges of keys stored in each subtree; if m_i is any key stored in the subtree with root $c_i, 1 \leq i \leq n$, then

$$m_1 \leq k_1 \leq m_2 \leq k_2 \leq \ldots \leq k_n \leq m_{n+1}$$

4. All leaves have the same depth, which is the tree's height $h(T)$.
5. Each internal node except the root contains at least $t-1$ and at most $2t-1$ keys. If tree is nonempty, then root has at least one key. Integer $t \geq 0$ is called node degree.
6. Every node x is read from an external memory by calling read(x) and written by calling write(x).

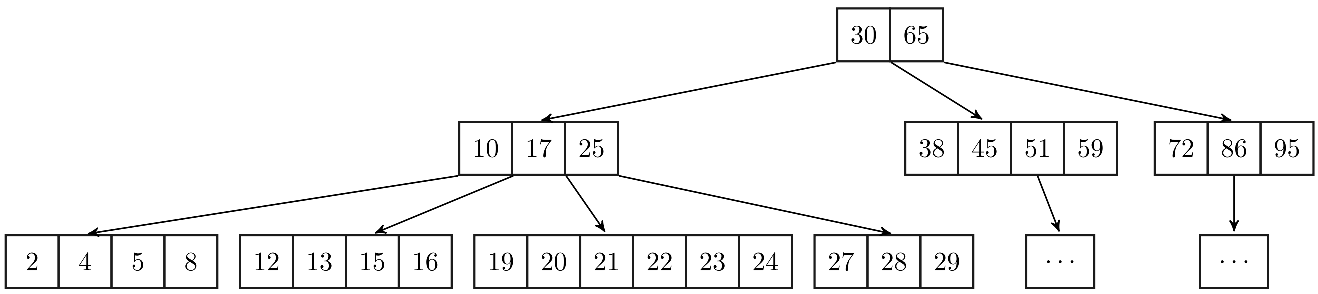

Figure 1: Example of B tree of degree $t = 4$

In the pseudo code, root of B tree T is denoted with $\mathsf{root}(T)$, degree of T with $\mathsf{degree}(T)$, height with $\mathsf{height}(T)$, number of stored keys in x with $\mathsf{keys\_no}(x)$, i-th child in x with $\mathsf{child}(i, x)$, i-th key of x with $\mathsf{key}(i, x)$

1.2 Searching

To find key K in a subtree at node x, the given node is checked for existence of such key. If not found, the correct subtree c_i is determined to check recursively. Adjacent keys k_i and k_{i+1} such that $k_i \leq K \leq k_{i+1}$ are found, then searching is continued on c_i.

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 $z \neq$ **null**
    $i = 1$
    while $i \leq$ keys_no(z) **and** $K >$ key(i, z)
      $i = i + 1$
    if $i \leq$ keys_no(z) **and** $K =$ key(i, 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

Finding index i at node x such that $K = k_i$ for the given key K is trivial.

Input: key K to find in node x of degree t
Output: index i of x or null
Complexity: $O(t)$
index(K, x)
  for $i = 1$ **to** keys_no(x)
    if $K =$ key(i, x)
      return i
  return null

Finding index i such that given key K fits into c_i's keys range is also trivial.

Input: key K to find a corresponding child in node x of degree t
Output: index i such that K belongs to $c_i(x)$ or null
Complexity: $O(t)$
index_child(K, x)
  for $i = 1$ **to** keys_no(x)
    if key$(i, x) \leq K \leq$ key$(i + 1, x)$
      return i
  return null

Finding predecessor key of the given k_i in node x is finding the right most key in the subtree c_i. Similarly, finding successor key of the given k_i in node x is finding the left most key in the subtree c_{i+1}.

Input: index i of the node $x \in T$, where T has n elements
Output: predecessor k_j of k_i determined as node y and index j; null if not found
Complexity: $O(\log n)$
predecessor(x, i)
  if leaf(x)
    $y = x$
    if $i = 1$
      $(y, j) =$ **null**
    else
      $j = i - 1$
  else

```
    x = read(child(i, x))
    while not leaf(x)
      x = read(child(keys_no(x) + 1, x))
    y = x, j = keys_no(x)
  return (y, j)
```

Input: index i of the node $x \in T$, where T has n elements
Output: successor k_j of k_i determined as node y and index j; null if not found
Complexity: $O(\log n)$

```
successor(x, i)
  if leaf(x)
    y = x
    if i = keys_no(x)
      (y, j) = null
    else
      j = i + 1
  else
    x = read(child(i + 1, x))
    while not leaf(x)
      x = read(child(1, x))
    y = x, j = 1
  return (y, j)
```

1.3 Auxiliary node operations

Splitting child node c_i of x is an operation performed on a full node c_i ($n = 2t - 1$ where n is number of keys in c_i) and x is not full. Splitting moves central key (the one at t-th place) to the correct place at the parent. The picture shows splitting node of seven keys to two nodes of three, while key **26** is moved up.

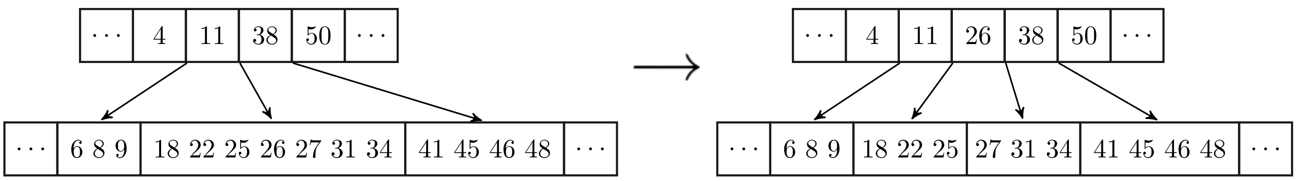

Figure 2: splitting 7-elements node ($t = 4$)

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}
  for j = 1 to degree(T) - 1
    key(j, z) = key(degree(T) + j, y)
  {copy second half of children from y to z}
  if not leaf(y)
```

```
    for j = 1 to degree(T)
      child(j, z) = child(degree(T) + 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
  write(x)
  write(y)
  write(z)
```

Merging is an operation reversed to the split operation. For a node x with at least t keys and children c_i and c_{i+1} with $t-1$ keys – the key k_i of x, all keys k_j of c_i and all keys k_l of c_{i+1} (where $1 \leq j, l \leq t-1$) are collapsed into single c_i node with $2t-1$ keys. The picture is analogous to the one of splitting node.

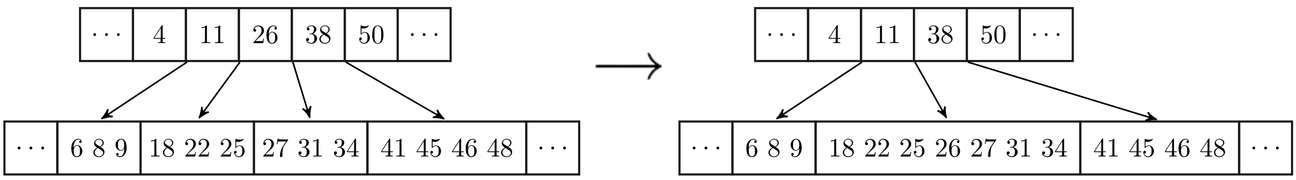

Figure 3: merging two 3-elements nodes ($t = 4$)

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 · degree(T) − 1
  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)
```

Key can be moved from node a (assuming that number of keys is not less than t) to immediate sibling b (assuming that number of keys is less than $2t-1$). Let x be their common parent, so $a = c_i$ and $b = c_{i+1}$ for some i; let p_j be the last key in a which is going to be moved. Since

$$K \le p_j \le k_i \le L \le k_{i+1} \text{ for all } K \in c_i, L \in c_{i+1}$$

p_j becomes the new k_i and old k_i becomes the first key q_1 in b. Old keys in b are moved one place to the right, as well b's children if b is not leaf. Also if a is not leaf, then it's child d_j can stay on it's own place but d_{j+1} has to be moved. Because new k_i has value of p_j and new q_1 has value of old k_i, without violating B tree properties it can be set $e_1 = d_{j+1}$ (e_1 is the first child in b). Since k_i is the only key affected by moving and $a = c_i$, $b = c_{i+1}$, no child of x is moved to the right. The picture shows moving of key **36**.

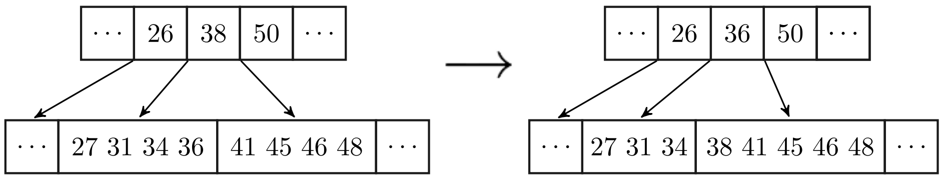

Figure 4: moving key 36

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(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)
```

Symetrically, first key from node $a = c_i$, $2 \leq i \leq n+1$, with the number of keys not less that t, can be moved to immediate sibling $b = c_{i-1}$, with the number of keys less that $2t-1$.

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(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 Inserting

Inserting key into B tree is about finding appropriate non-full leaf node to insert the key. To insert key K into non-full node x, check if x is leaf – if does, find the right place to insert; if not, then insert into a child where K belongs.

Input: key K to insert into non-full node $x \in T$, where T has n nodes
Output: none
Complexity: $O(\log n)$

```
insert(x, K)
  i = keys_no(x)
  if leaf(x)
    {inserting into leaf is putting key to the proper position}
    while i ≥ 1 and K < key(i, x)
      key(i + 1, x) = key(i, x)
      i = i − 1
    key(i + 1, x) = K
    keys_no(x) = keys_no(x) + 1
    write(x)
  else
    while i ≥ 1 and K < key(i, x)
      i = i − 1
    i = i + 1
    read(child(i, x))
```

```
    if keys_no(child(i, x)) = 2 · degree(T) − 1
      split(x, i)
      {key from c_i moved up to x, so check if K should be moved too}
      if K > key(i, x)
        i = i + 1
    insert(child(i, x), K)
```

To insert key K into tree T, the algorithm starts at the root. If root is not full, use the above insert function directly. If not, create new root and split the original root.

Input: key K to insert into T with n elements
Output: none
Complexity: $O(\log n)$

```
insert(K)
  if keys_no(root(T)) = 2 · degree(T) − 1
    new s
    root(T) = s
    leaf(s) = false
    keys_no(s) = 0
    child(1, s) = root(T)
    split(s, 1)
    insert(s, K)
  else
    insert(root(T), K)
```

1.5 Deleting

Deleting distinguishes cases on leaves and internal nodes. The following situations are possible for key K and subtree x:

D1 If the key K is in leaf x, then delete the key K from x.
D2 If the key K is in internal node x, then:

- **D2.1** If x's child y that precedes K has at least t keys, then delete the predecessor K' (which is placed in leaf of subtree y) of K and replace K by K' in x.
- **D2.2** Symmetrically, if x's child z that follows K has at least t keys, then delete the successor K' (which is stored in leaf of subtree z) of K and replace K by K' in x.
- **D2.3** Otherwise, if both y and z have only $t-1$ keys, merge K and all of z into y, so that x loses both K and the pointer to z, and y now contains $2t-1$ keys. Then, delete z and recursively delete K from y.

D3 If the key K is not present in internal node x, find child c_i that contains K. If c_i has only $t-1$ keys, execute step D3.1 or D3.2 as necessary to guarantee that we descend to a node containing at least t keys. Then, recursively delete K on c_i.

- **D3.1** 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.
- **D3.2** 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 {cases D1 - D2}
    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}
      y = child(i, x), z = child(i + 1, x)
      if keys_no(y) ≥ t {case D2.1}
        (a, j) = predecessor(x, i)
        K′ = key(j, a)
        delete(y, K′) {case D1}
        key(i, x) = K′
        write(x)
      else if keys_no(z) ≥ t {case D2.2}
        (a, j) = successor(x, i)
        K′ = key(j, a)
        delete(z, K′) {case D1}
        key(i, x) = K′
        write(x)
      else {case D2.3}
        merge(x, i) {moves K from x to y}
        delete(y, K) {case D3}
  else {case D3}
    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 D3.1}
          move_key_next(x, i − 1)
        else if keys_no(child(i + 1, x)) ≥ t {case D3.1}
          move_key_prev(x, i + 1)
        else {case D3.2}
          merge(x, i)
      else if i = 1
        if keys_no(child(i + 1, x)) = degree(T) − 1 {case D3.2}
          merge(x, i)
        else {case D3.1}
          mode_key_prev(x, i + 1)
      else if i = keys_no(x) + 1
        if keys_no(child(i − 1, x)) = degree(T) − 1 {case D3.2}
          merge(x, i − 1)
        else {case D3.1}
          move_key_next(x, i − 1)
      delete(child(i, x), K)
    else
      delete(child(i, x), K)
  return x
```

1.6 Worst case complexity

B tree with one, two or three elements has only one (root) node. B tree with four elements can have at most two nodes, having at least two elements in the child element. If node x has zero keys then it has one child.

Lemma 1.1. If $n \geq 1$ and $t \geq 2$, then for every tree with n nodes and degree t, height of the tree is not greater than $\log_t \frac{n+1}{2}$.

Theorem 1.2. Complexity of find, insert and delete operations is $O(\log n)$.

Proof Follows from lemma 1.1. **QED**

1.7 Notes

B tree* is a B tree where each node has at least $\frac{2}{3}$ full, i.e. contains at least $\frac{4}{3}t - 1$ keys. Inserting splits two full sibling nodes into three, so each of them is $\frac{2}{3}$ full. Since this scheme ensures that storage utilization is relatively high, height of B* tree is relatively smaller, consequently the find operation takes less time than in B tree.

Red black tree where each black node absorbs its red children is B tree. Such black node becomes node with three keys and four children at most.

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