2-3 Trees with Recursive Algorithms for Elementary Opertaions

Coordinator: Mihaescu Cristian,Phd
Student: Stefan Cristian Mladin

University of Craiova - Faculty of Automatics, Computers and Electronics
Computer Engineering

DCTI IT Companies Seminary, April 2014

Definition

A *2-3 search tree* t is either:

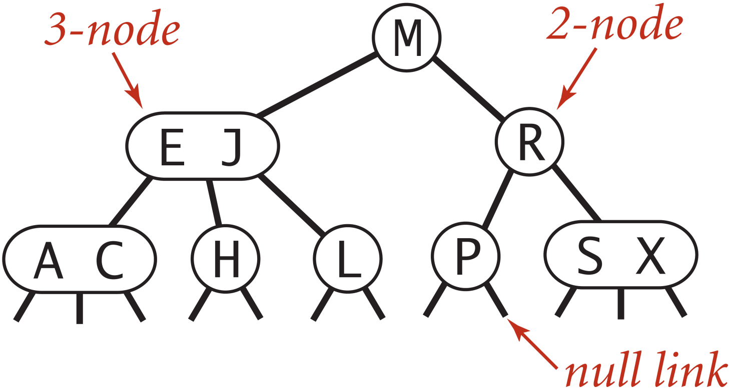
internal 2 node: $K, [t_l, t_r]$ where t_l,t_r are 2-3 trees, every key in t_l is lesser than K, every key in t_r is greater than K.

internal 3 node: $[K_1, K_2], [t_l, t_m, t_r]$ where t_l,t_m,t_r are 2-3 trees,$K_2 > K_1$, every key in t_l is lesser than K_1, every key in t_m is greater than K_1 and lesser than K_2, every key in t_r is greater than K_2.

leaf 2 node a 2 node with one key and an empty tree list.

leaf 3 node a 3 node with 2 keys:K_1, K_2 and empty tree list where $K_2 > K_1$.

[4, 3, 1, 2]


3-node
2-node
M
E J
R
A C
H
L
P
S X
null link

Data: A 2-3 tree t and a key K
Result: The node in the tree t containing the key K or 0(NULL)

if $K = K_1$ *or* $K = K_2$ **then**
  ▷RETURN t;
else
  if t *leaf* **then**
    ▷RETURN 0;
  else
    ▷RETURN $search(next(t, K), K)$;
  end
end

Procedure search(t,K)

The worst case scenario is when the key that is being searched for is located at leaf level.
Because a 2-3 tree has a height between $log_3 N$ and $log_2 N$ the complexity of the search operation in a 2-3 tree is $O(\log N)$.
Since insertion and deletion are done at leaf level they will also have a complexity of $O(\log N)$.

Algorithm for Key Insertion

```
Data: t a node from the insert path. K the key.
Result: Returns 0 if t is not root, or the root of the tree resulting
        from inserting K, if t is root.
if ▷t is empty tree then
  │ ▷ RETURN created tree with the value K ; Empty tree case
end
if ▷t is leaf then
  │ LEAF PHASE: recursion to the apropriate leaf
  │ t′ ← push(t, K);
  │ if ▷t′ is 4 node then
  │   │ excessSplit ← split(t′);
  │   │ excessInsert(t, K) ← excessSplit;   (4)leaf 3 node case
  │ else
  │   │ t ← t′;
  │   │ excessInsert(t, K) ← 0;             (1)leaf 2 node case
  │ end
else
  │ DOWNWARD PHASE: next slide
```

```
if ▷t is leaf then
  LEAF PHASE: see previous slide
else
  DOWNWARD PHASE: recursing downwards towards leaf
  INSERT(next(t, K), K);
  UPWARDS PHASE: adding the excess from the lower
  levels and computing the excess for the upper
  levels
  if excessInsert(next(t, K), K) = 0) then
    excessInsert(t, K) ← 0;    (2)no excess received case
  else
    t' ← push(t, excessInsert(next(t, K), K));
    if ▷t' is 3 node then
      t ← t';
      excessInsert(t, K) ← 0;    (3)upwards excess stop
      case
    else
      (5)upwards excess continuation case:next
      slide
    end
```

if $\triangleright$ *t is leaf* **then**

```
LEAF PHASE: see previous slide
```

else

```
DOWNWARD PHASE: see previous slide
UPWARDS PHASE
```

if *excessInsert*$(next(t, K), K) = 0)$ **then**

```
(2)no excess received case:  see previous slide
```

else

if $\triangleright t'$ *is 3 node* **then**

```
(3)upwards excess stop case:  previous slide
```

else

excessSplit $\leftarrow$ *split*(t');

excessInsert$(t, K) \leftarrow$ *excessSplit*; `(5)upwards excess continuation case`

end

end

end

if $\triangleright$ *t is root* **then**

```
Reaching the root:  next slide.
```

```
if ▷ t is root then
    Reaching the root
    if excessInsert(t, K) = 0 then
        ▷ RETURN t ;    There was no split in the root.
        The root remains the same.
    else
        ▷ RETURN excessInsert(t, k) ;  There was a split in
        the root.  The new root is the split result of
        the current one.
    end
else
    ▷ RETURN 0;
end
```

Procedure INSERT(t,K)

Algorithm for Key Deletion

The algorithm for key deletion here.

Insertion Example

In the following example the insertion algorithm and it's cases will be illustrated by inserting the keys $1, 2, 3 \ldots 7$ in that order, in an empty 2-3 tree.
When inserting 1 the insertion is called once. It's in the empty tree case.

Call Stack:

$insert\_rec(t = 0x00000000, K = 1, excessInsert = 0x00000000)$

$\boxed{1}$

When inserting 2 the insertion is called. It's the 2 leaf case non-excess generating case. Since the tree is non empty the key is pushed in the root.

Call Stack:

$insert\_rec(t = 0x005f0660, K = 2, excessInsert = 0x00000000)$
$pushSorted(t = 0x005f0660, tK, loc)$

1	2

When inserting 3 the insertion is called once. It's the 3 leaf case excess generating case. The excess is at root level so the tree increases it's level with 1.

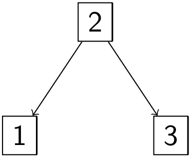
Call Stack:

$insert\_rec(t = 0x005f0660, K = 3, excessInsert = 0x00000000)$

$pushSorted(t = 0x005f0660, tK, loc = 0x005f2ce0)$

1	2	3

$split(t = 0x005f2ce0, loc = 0x005f2d88)$



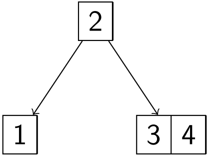
Since the tree has 2 levels now the insertion function is called twice for inserting 4. It's put in a leaf 2 node.

Call Stack:

$insert\_rec(t = 0x005f2d88, K = 4, excessInsert = 0x00000000)$

$insert\_rec(t = 0x005f2e28, K = 4, excessInsert = 0x0041f808)$

$pushSorted(t = 0x005f2e28, tK, loc)$



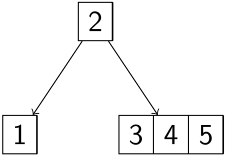
Since the tree has 2 levels now the insertion function is called twice for inserting 5. It's put in a leaf 3 node so the split function is called. The excess is then pushed in the root.

Call Stack:

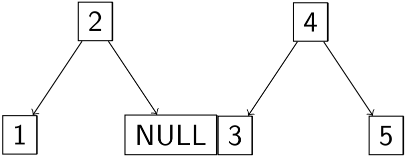
$insert\_rec(t = 0x005f2d88, K = 5, excessInsert = 0x00000000)$

$insert\_rec(t = 0x005f2e28, K = 5, excessInsert = 0x0041f808)$

$pushSorted(t = 0x005f2e28, tK, loc = 0x005f2d30)$



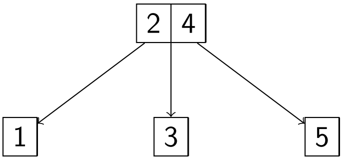
$split(t = 0x005f2d30, loc = 0x005f2ce0)$



Returning to the first insertion call in the call stack where $excessInsert = 0x0041f808$ was allocated for the second call:

Call Stack:

$insert\_rec(t = 0x005f2d88, K = 5, excessInsert = 0x00000000)$
$pushSorted(t = 0x005f2d88, tK = (0x0041f808) = 0x005f2ce0$[1], $loc = 0x005f2d30)$

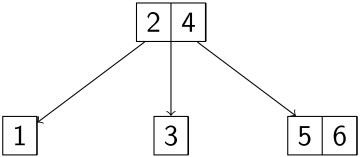


[1] Note that (0x0041f808) = 0x005f2ce0. In the C implementation excessInsert is a pointer to a 2-3 tree pointer.

When inserting 6 the insertion is called 2 times to the appropriate leaf where the key is added.

Call Stack:

$insert\_rec(t = 0x005f2d88, K = 6, excessInsert = 0x00000000)$
$insert\_rec(t = 0x005f2f18, K = 6, excessInsert = 0x0041f808)$
$pushSorted(t = 0x005f2f18, tK, loc)$



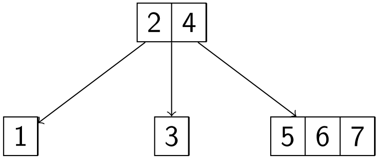
When inserting 7 the insertion is called 2 times to the appropriate leaf where the key is added. Since it's a 3 leaf node a split is required pushing excess to the first call of the insertion function.

Call Stack:

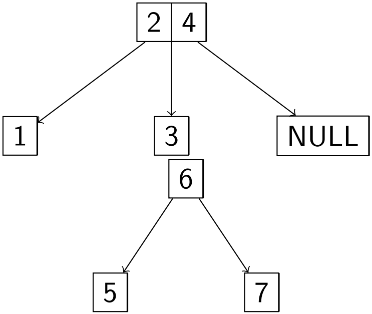
$insert\_rec(t = 0x005f2d88, K = 7, excessInsert = 0x00000000)$

$insert\_rec(t = 0x005f2f18, K = 7, excessInsert = 0x0041f808)$

$pushSorted(t = 0x005f2f18, tK, loc = 0x005f2ce0)$



$split(t = 0x005f2ce0, loc = 0x005f2e78)$



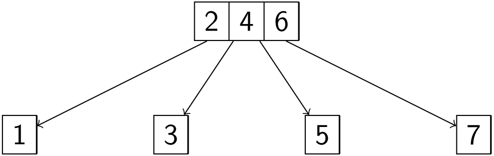
Returning to the first insertion call in the call stack where $excessInsert = 0x0041f808$ was allocated for the second call:

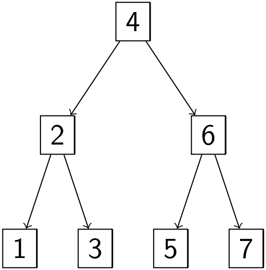
Call Stack:

$insert\_rec(t = 0x005f2d88, K = 7, excessInsert = 0x00000000)$

$pushSorted(t = 0x005f2f18, tK = (0x0041f808) = 0x005f2e78, loc = 0x005f2ce0)$

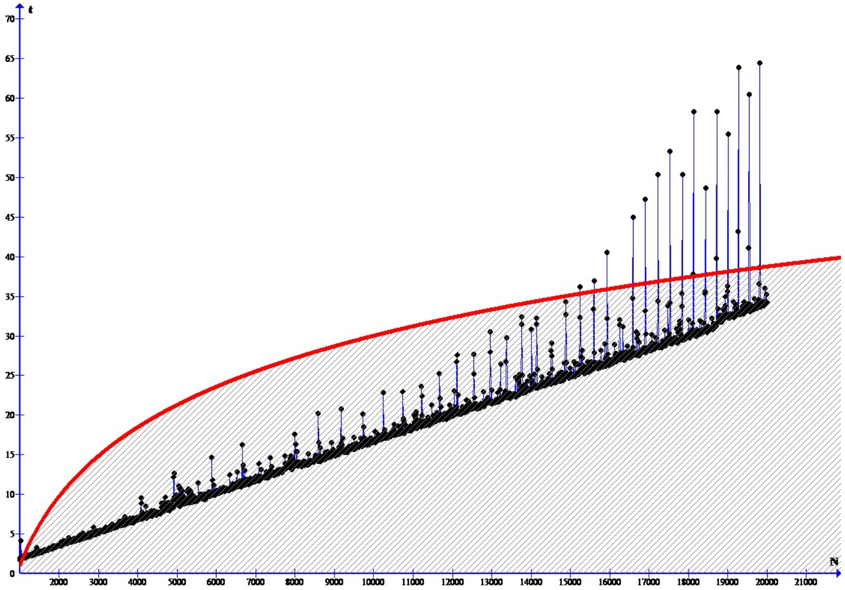
$split(t = 0x005f2ce0, loc = 0x005f2d38)$





Since the split was done at root level, the tree will increase height by 1 with the new root being the result of the split.

Experimental Results



A series of tests were generated each consisting in creating a tree with a number of nodes between 1000 and 20000. For each test the time it took to create the tree was measured and plotted.

References

- Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
  Data Structures and Algorithms.
  11 edition.
- Dumitru Dan Burdescu and Marian Cristian Mihaescu.
  Algorithms and Data Structures.
- Robert Sedgewick and Kevin Wayne.
  Algorithms.
  Addison-Wesley, 4 edition, March 2011.
- Wikipedia.
  Tree (data structure) — wikipedia, the free encyclopedia, 2013.
  http://en.wikipedia.org/w/index.php?title=Tree_(data_structure)&oldid=586195126.