

SEARCH TREES OF BOUNDED DEPTH/HEIGHT

(2,3,4)-trees, (a,b)-trees, B-trees

Chapter 20-20.3 of GT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.*
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis

Search trees of bounded height

Introduction

In databases, binary search trees (balanced or unbalanced) is not the most efficient way to store information.

If every node of the tree is stored in a separate sector or cluster or track or block of secondary memory (e.g. an external memory hard disk drive), a significant segment of bytes is read from secondary memory to access a single record for a node x of the binary search tree.

The information associated with x consists of only few bytes (for a left, right subtree references, a parent reference and a key value).

This is too (secondary) memory communication inefficient. Bear in mind accessing even one byte of sector from an external hard disk drive which is secondary memory takes several, if not tens of, milliseconds.

Accessing that byte from the main memory takes tens of nanoseconds, that is it is tens of thousands if not hundreds of thousands faster.

For such cases other methods of organizing a tree structure can be used.

In such structures more information (key values) are stored at a node. Thus whereas in binary search tree the number of keys is equal to the number of internal nodes (one key per node), and the leaves are external nodes and one more than the number of internal nodes, this is not the case any more for other alternative methods of organization.

Definition 1 (Bounded Height Trees). A bounded height tree is a tree whose height is bounded.

Definition 2 (Bounded Depth Trees). A bounded depth tree is a tree whose depth is bounded.

Search trees of bounded height

Introduction

Note that the aforementioned definitions are too general and omit some implied details.

When we discuss height we discuss it in the context of a rooted tree. Some better or more complete definitions follow.

Definition 3 (Bounded Height Search Trees). A bounded height (binary) search tree is a search tree whose height is bounded (usually) by the logarithm of the number of keys n maintained in the tree.

Definition 4 (Bounded Depth Search Trees). A bounded depth (binary) search tree is a search tree whose height is bounded (usually) by the logarithm of the number of keys n maintained in the tree.

DRAFT

Search trees of bounded height

Introduction continues

The definitions selectively omitted the word binary for two reasons: (a) it is implied, (b) so that it can be used for trees that are not binary trees.

Example 1. An AVL tree T maintaining n keys is a bounded height (binary) search tree. This is because $h = \Theta(\lg n)$ where n is the number of keys (values) in T . Moreover n is also the number of internal nodes in T , or $n + 1$ is the number of external nodes (leaves) in T .

Example 2. A red-black tree T maintaining n keys is a bounded height (binary) search tree. This is because $h = \Theta(\lg n)$ where n is the number of keys (values) in T .

Proposition 1. *If a binary search tree is of bounded height then it is also of bounded depth.*

In the next section we are going to discuss search trees of bounded depth. Such trees are also going to be of bounded height as well.

However they are going to exhibit several differences compared to binary search trees. They are not going to be binary for a start. Moreover, the bounded depth property would be defined differently: (a) the binary search tree property would be modified accordingly to reflect that they are not binary trees, (b) all leaves (also known as external nodes) are going to be of the same depth, something that was not (necessarily) the cases for (balanced) binary search trees as as the AVL trees that we discussed in detail or the red-black trees that we did not discuss in detail.

One such way is a generalization of binary search trees known as m -way search trees or multi-way search trees.

Multi-way search trees

Introduction

In a binary search tree every node has one or zero key values. If the number of key values at a node is zero, we call such a node a leaf or an external node and we represented it with a NULL. Otherwise a node is associated with one key value. In that latter case we call the node an internal node. Then that node has two children with 0, 1 or 2 key values.

For a multi-way search tree T also known as an m -way search tree T or just MWST T , a node v of T is a q -node, if v has q children, where $q \leq m$.

Definition 5 (q -node). A node in a search tree is a q -node if it has q children.

Corollary 1. Every node of a binary search tree is a 2-node.

Corollary 2. Every q -node contains $q - 1$ key values.

Corollary 3. Every 2-node of binary search tree has one key value.

Note that the NULL nodes are counted as children in addition to nodes with key values.

The search tree property of an MWST is known as the MWSTP (Multi-way search tree property) and generalizes directly the BSTP (Binary search tree property) of a BST. Moreover, a MWST T is also a rooted and ordered tree. Note that the terms items keys and values would be used interchangeably. An item contains a key plus some auxiliary information. A key has a value that identifies the key.

Definition 6 (Multi-way search tree or m -way search tree). An m -way search tree (MWST) T is defined as follows.

- An empty tree is a MWST,
- each internal node of T has at least two children,
- each internal node stores a collection of items of the form (k, x) , where k is a key value and x is some information related to k (such information x we shall ignore in the remainder),
- a node of T contains at most $m - 1$ items and has at most m children,
- if $(k_1, x_1), \dots, (k_{d-1}, x_{d-1})$ are the $d - 1$ items of a node w ordered so that $k_1 < k_2 < \dots < k_{d-1}$, and if $c_0, \dots, c_{d-1}$ are the d children of w , $d \leq m$, then the keys in the subtree rooted at c_0 are less than k_1 , the keys in the subtree rooted at c_d are greater than k_d , and the keys in the subtree rooted at c_i , $0 < i \leq d$, are between k_{i-1} and k_i thus establishing the MWSTP (multi-way search tree property), and finally
- NULL pointers are represented as special external nodes with no children and no items.

Lemma 2 (MWST). *In a MWST T every internal node has at most m children and thus every internal node has at most $\leq m - 1$ key values.*

A BST tree can be viewed as an MWST with $m = 2$.

Minimal and Maximal

Minimal and Maximal *m*-way trees. The minimal *m*-way tree of height *h* consists of *h* internal nodes, one internal node per level, each node containing one item (think of a linked list extending to the right/bottom or the left/bottom). Thus the minimal number of items in an *m*-way tree of height *h* is *h*. In a maximal *m*-way tree of height *h*, every internal node of level $i = 1, \dots, h - 1$ has *m* children. Then, the total number of internal nodes is $1 + m + \dots + m^{h-1} = (m^h - 1)/(m - 1)$. Each node contains *m* - 1 items and thus *m* - 1 key values. Therefore the total number of items in such a tree is $m^h - 1$. Thus the height of an *m*-way tree of *n* items can vary from *n* all the way down to $\log_m(n + 1)$. The latter logarithm is base *m*.

DRAFT

Balancing m -way (search) trees

In order to guarantee that an m -way tree is balanced and by extension it is of bounded height and also of bounded depth (and this would mean height is in the order of $\log_m(n+1)$), we need to impose upon special properties on its structure that would guarantee their balancing.

Balanced m -way tree properties.

All such variants share the same important property. A node can not have very few items / key values as it is the case for a minimal m -way tree for example. In all three variants the minimum number of children per node is about half the maximum.

This property alone guarantees that the height of such an m -way tree is close to $\log_m(n+1)$ i.e. $\Theta(\log_m(n+1))$.

We discuss balanced search trees and their operations in more detail in the next section. All such trees are balanced m -way trees but the value of m might be restricted in certain way to guarantee the balancing property.

DRAFT

Example 3.

UNBALANCED SEARCH TREES

m-way search tree

Every node has $\leq m$ children

Binary search tree is a

2-way tree

(a,b)-tree

Every node has $\leq b$ children

Every node has $\geq a$ children

(2,4)-tree aka (2,3,4)-tree

Every node has ≤ 4 children

Every node has ≥ 2 children

BALANCED SEARCH TREES

P1 : root allowed to have
2 children

P2 : all NULL at the same
bottom level/depth

(a,b)-tree $2 \leq a \leq (b+1)/2$

B-tree = (a,b)-tree with

$a = \text{ceil}(B/2)$

$b = B$

2-3-4 tree is a B-tree: $B=4$

Balanced search trees

Preliminary definitions

In order to have balanced search trees maintaining an arbitrary non-negative number of keys values two properties must be defined and maintained before and after a modifying operation (e.g. insert and delete).

The first one deals with sparsely populated trees.

Property 1 (Root property). A root in a (balanced) search tree is allowed to have just two children without violating other properties.

Property 2 (Bounded-depth property). All NULL (external nodes) in a (balanced) search tree are at the same level (e.g. depth or height) of the search tree.

With these two properties various subclasses of multi-way search trees becomes balanced.

DRAFT

(a, b) -trees Definitions

An (a, b) -tree T is a search tree and in fact it is a MWST. An (a, b) -tree is a restricted form of an MWST defined as follows.

Definition 7 ((a, b) -tree). An (a, b) -tree T is a multi-way search tree where every internal node has at most $\leq b$ children and at least $\geq a$ children, where $a \leq b$, and thus every internal node has at most $\leq b - 1$ key values and at least $\geq a - 1$ key values. Moreover the root property and bounded-depth property are maintained and in addition, $2 \leq a \leq (b + 1)/2$.

Proposition 3 ((a, b) -tree is of bounded depth and height). *An (a, b) -tree as previously defined is of bounded depth.*

Proof. The proof is deferred until Theorem 1 and Theorem 2 establish an upper bound and a lower bound on the height of an (a, b) -tree. ■

Definition 8 $((2, 4)$ -tree also known as $(2, 3, 4)$ -tree). A $(2, 4)$ -tree is an (a, b) -tree with $a = 2$ and $b = 4$.

In other words every internal node has ≥ 2 and ≤ 4 children, or equivalently 2, 3, or 4 children.

Definition 9 (B -tree). A B -tree is an (a, b) -tree with $a = \lceil B/2 \rceil$ and $b = B$.

Lemma 4. *A $(2, 4)$ -tree is a B -tree.*

Lemma 5. *In an (a, b) -tree let n be the number of keys maintained by it. Then the number of NULL nodes (external nodes) is $n + 1$.*

Note that the number of internal nodes is usually less than n since a node can store multiple key values.

DRAFT

Height and Depth properties

We now establish the components of the deferred proof of Proposition 3.

Theorem 1. *A minimal (a, b) -tree has height $h = O(\lg n / \lg a)$.*

Proof. In a minimal (a, b) -tree the root node has the minimal number of children (i.e. 2) and every other node has the minimal prescribed number of children (i.e. a). For a tree of height h the external nodes are at the bottom and at level h . Thus the number of nodes at level 0 is 1, the number of nodes at level 1 is 2, the number of nodes at level 2 is $2a$, and the number of nodes at level i is $2a^{i-1}$. Thus the total number of nodes at level h is $2a^{h-1}$. The number of external nodes (NULL) is $n + 1$ and must be $n + 1 \geq 2a^{h-1}$ and thus

$$h \leq \frac{\lg(n + 1) - 1}{\lg a} + 1.$$

■

Theorem 2. *A maximal (a, b) -tree has height $h = \Omega(\lg n / \lg b)$.*

Proof. In a maximal (a, b) -tree the root node has the maximal number of children (i.e. b) and every other node has also the maximal prescribed number of children (i.e. b). For a tree of height h the external nodes are at the bottom and at level h . Thus the number of nodes at level 0 is 1, the number of nodes at level 1 is b , the number of nodes at level 2 is b^2 , and the number of nodes at level i is b^i . Thus the total number of nodes at level h is b^h . The number of external nodes (NULL) is $n + 1$ and must be accommodated within the bounds of this maximal tree, $n + 1 \leq b^h$ and thus

$$h \geq \frac{\lg(n + 1)}{\lg b}.$$

■

Balanced search trees

Preliminary definitions

Example 4.

(a,b) -tree $2 \leq a \leq (b+1)/2$ $\text{numOfkeys} = n$; $\text{numOfNULL} = n+1$

MINIMAL- (a,b) tree
 Root has minimally 2 children
 Every other node a children

	Level	NumOf Nodes
x root	0	1
x x	1	2
x x	2	$2a$
x x		
x x	i	$2a^{**i-1}$
x x		
x x	h	$2a^{**h-1} \leq n+1$
<- $n+1$ NULL ->		

MAXIMAL (a,b) -tree
 Root has maximally b children
 Every other node has b children

	Level	NumOfNodes
x root	0	1
x x	1	b
x x	2	b^{**2}
x x		
x x	i	b^{**i}
x x		
x x	h	$b^{**h} \geq n+1$
<- $n+1$ NULL->		

Lemma 6. *The height of an (a, b) -tree with n keys is*

$$\frac{\lg(n+1)}{\lg b} \leq h \leq \frac{\lg(n+1) - 1}{\lg a} + 1.$$

Proof. Immediately follows from the prior discussion. ■

DRAFT

$(2, 3, 4)$ -trees and B -trees

Height and Depth properties

Lemma 7. *The height of a 2-3-4 tree with n keys is $\Theta(\lg n)$.*

Proof. Set $a = 2$ and $b = 4$ in Theorem 1 and Theorem 2 or Lemma 6 and obtain the following.

$$\frac{\lg(n+1)}{\lg 4} \leq h \leq \frac{\lg(n+1) - 1}{\lg 2} + 1.$$

Thus

$$\frac{\lg(n+1)}{2} \leq h \leq \lg(n+1).$$

The result follows. ■

Lemma 8. *The height of a B -tree with n keys is $\Theta(\lg n / \lg B)$.*

Proof. Set $a = \lceil B/2 \rceil$ and $b = B$, in Theorem 1 and Theorem 2 or Lemma 6 and obtain the following.

$$\frac{\lg(n+1)}{\lg B} \leq h \leq \frac{\lg(n+1) - 1}{\lg(\lceil B/2 \rceil)} + 1.$$

The result follows. ■

In the following section, we discuss operations on the variety of (a, b) -trees introduced earlier. The discussion is tree agnostic. Thus we prefer to use an m to indicate the a or more frequently the b of an (a, b) -tree, or the B of a B -tree or the 4 of a $(2, 3, 4)$ -tree.

Bounded depth search trees

Probing operations

We give a very rough outline of how the generic operations for m -way trees are performed. We describe explicitly the operations for $2-3-4$, where $m = 4$, (a, b) -tree with $m = b$, and B trees, where $m = B$.

Proposition 9. *All BST probing operations described in the previous chapter work in a similar way on an (a, b) -tree, $(2, 3, 4)$ -tree or B -tree. The running time of the operations, in terms of keys comparisons, is $O(\lg n)$ if n is the number of keys in the tree.*

Proof. We describe operations in an agnostic way. Thus m below should be considered to be $m = b$ for an (a, b) -tree, $m = 4$ for a $(2, 3, 4)$ -tree and $m = B$ for a B -tree. We avoid using a prefix such as BST or AVL to describe the relevant operation. Let T be an instance of a search tree of bounded depth (and also height). A **Search**(k, r) operation initiates a search traversal at the root r of T for key k . It will eventually generate either a path to a node that contains the key value k or to an external node that is a NULL. In that path, if a node w was encountered with d items, i.e. the original **Search**(k, r) has become a **Search**(k, w), k would be compared to the d keys k_i stored in the items of w . With the k_i appearing in sorted order, by using binary search in $O(\lg((d-1)+1)) = O(\lg d)$ comparisons, and for $d \leq m$ subsequently in $O(\lg m)$ comparisons we conclude one of the following:

Case a. There exists an i such that $k_i = k$. The key has been found (successful search). The search process terminates.

Case b. There exists no i such that $k_i = k$. The key has not been found (unsuccessful search) at node w . Yet we have determined that k is between k_i and k_{i+1} , or $k < k_1$, or $k > k_{d-1}$. Let c_i be the $i+1$ -st child of w . We then recur by following one of c_i , c_0 or c_{d-1} recursively. Thus the search operation **Search**(k, w) has been reduced to **Search**(k, c_i) or **Search**(k, c_0) or **Search**(k, c_{d-1}). If one of the three candidates c_* is a NULL the search process has reached a leaf (external node). The **Search** terminates unsuccessfully. (The star $*$ is a wildcard character that indicates based on the case an i , or a 0 or a $d-1$ as needed.)

All other probing operations such as **Min**, **Max**, **Predecessor**, and **Successor** are performed similarly. The running time at every level requires $O(\lg m)$ comparisons. The height of the tree is $O(\lg n / \lg m)$. Multiplying the two we obtain a $O(\lg n)$ running time bound reflecting comparisons performed.

Note: In databases and other setting we are more interested in the $O(\lg n / \lg m)$ factor as this factor determines the number of I/O operations or disk sector (cluster, track, block) operations performed. Such an operation takes milliseconds, substantially more than the nanoseconds of comparing to keys in main memory. Thus we could also add the number of nodes accessed in a **Search** operation is $O(\lg n / \lg m)$. ■

Bounded depth search trees

Insertion operation

Proposition 10 (Insertion). *Insertion of a key on an (a, b) -tree, $(2, 3, 4)$ -tree or B -tree works similarly to the `BSTInsert` or `AVLInsert` operation in the previous chapter except that the operation is completed by a `Fix` similarly to `AVLInsert`. The running time of `Insert`, in terms of number of comparisons performed, is $O(\lg n)$ if n is the number of keys in the tree.*

Proof. For a tree T we describe the `Insert(k, r)` in an agnostic way. The root of T is r and k is the key value that will be inserted into T . We assume we have no duplicates (or equivalently a successful search would abort the `Insert` operation). Thus m below should be considered to be $m = b$ for an (a, b) -tree, $m = 4$ for a $(2, 3, 4)$ -tree and $m = B$ for a B -tree. We avoid using a prefix such as `BST` or `AVL` to describe the relevant operation. Let T be an instance of a search tree of bounded depth (and also height). Just like a `BSTInsert` is similar to an unsuccessful `BSTSearch`, for an `Insert` operation a `Search` is realized that eventually will prove unsuccessful. Thus connecting with `Search` we have reached an external node (a leaf) previously referred to as node c_* . The key k is then inserted into the parent of c_* denoted in `Search` operation description as w . Thus we issue `InsertKey(k, parent(c_*))` which is in fact `InsertKey(k, parent(w))`. Thus a new item is created between c_i and c_{i+1} (we ignore the boundary case of c_0 and c_{d-1} that can be separately considered) with the new item containing (k, c_{i+1}) . At this point there are two possibilities:

Case a (Overflow leads to Splitting and InsertKey). The node contained the full complement of keys/items prior to the `Insertion` (at most $m - 1$ items) and currently contains m . This is known as **overflow**. The node is split into two nodes using operation `Split` around the upper median of the keys values. The original node w would be split into two new nodes w_l and w_r , one node w_l containing items with keys less than the upper median, the other node w_r containing items with keys greater than the upper median with the upper median key/item being inserted into the parent of w which would also become the parent of w_l and w_r using operation `InsertKey(UpperMedian, parent(w))`. The `InsertKey` operation is in fact the previously mentioned `Fix` operation that after a split can result into subsequent `Fix` / `InsertKey` higher in the tree. The process ends when the root of the tree is encountered. A `InsertKey(SomeKeyValue, r)` is possible to lead to a split at the root. This is easily fixed or dealt with: a root can be split into two nodes W_l and W_r similarly to previous split operations with a new root increasing the height of the tree by one. Note that the new root would have two children. The Root Property of balanced search trees takes care of the details.

Case b. There is no overflow at w and `Insertion` terminates at w successfully.

Bounded depth search trees

Insertion operation

Proof continued. Therefore Insertion might take time proportional to the height of the tree not only during the traversal that is similar to Search, but also because of InsertKey when it moves upwards and possibly all the way to the root. In terms of number of comparisons and other operations performed Insert is $O(h \times O(\lg m))$ which is $O(\lg n)$ just like the probing operations. ■

DRAFT

$(2, 3, 4)$ -tree Insert

Example 5 ($(2, 3, 4)$ -tree Insert). The pseudocode for $(2, 3, 4)$ -tree Insert follows. (This Insert with $m = 4$.)

Proof.

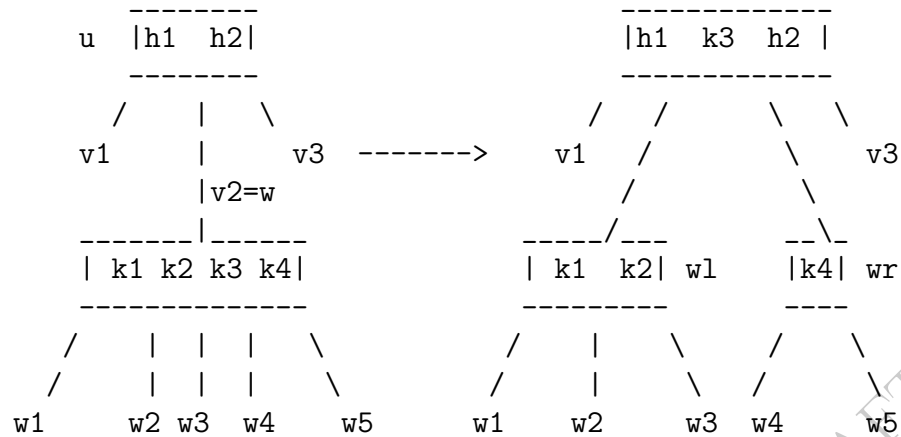
```
2-3-4Insert((k,x),root(T))    // Insert (k,x) in tree T.
1. if Search (k,root(T)) == Found    // root(T) is r
2.   Do-Nothing-and-Return();
3. else
4.   An external node z has been reached; let w be the parent of z.
5.   Add k into w and add a new child c (external node) to the left of z,
6.   i.e. add item (k,x,c) in w's data;
7.   if (w becomes a 5-node) {        // Overflow @ w
8.     Let w1,w2,w3,w4,w5 be the five children of w
9.     Let k1,k2,k3,k4 be the four keys stored in w.
10.    SplitAt(w);                    // Fix overflow with a Split at w

    SplitAt(w)
1. Find upper median k3.
2. Identify parent v of w; if w was root(T), create new root and call it v.
3. InsertKey(k3,v) (or k3 is the only key of new v=root(T))
4. Create w1 and w2 to replace w
5.   w1 has k1 k2 and w1 w2 w3
6.   w2 has k4 and w4 w5
7. If ( v != root(T)) && ( v is a 5-node)
8.   SplitAt(v)                      //Recur at parent v of w
9. else
10.  return();
```

The case depiction is on the next page.

$(2, 3, 4)$ -tree
Insert

Proof continues.

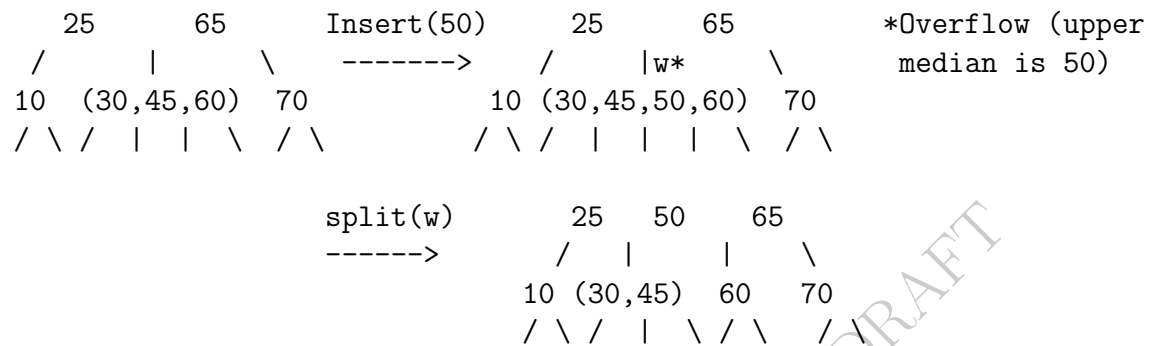


■

(2, 3, 4)-tree Example

Example 6. An example of Insert involving key with value 50 in a (2, 3, 4)-tree is shown.

Proof.



■

Bounded depth search trees

Deletion operation

Proposition 11 (Deletion). *Deletion on an (a, b) -tree, $(2, 3, 4)$ -tree or B -tree works similarly to the `BSTDelete` or `AVLDelete` operation in the previous chapter except that the operation is completed by a `Fix` similarly to `AVLDelete`. The running time of `Delete`, in terms of number of comparisons performed, is $O(\lg n)$ if n is the number of keys in the tree.*

Proof. For a tree T we describe the `Delete(k, r)` in an agnostic way. The root of T is r and z is the key value that will be deleted from T . We assume we have no duplicates (or equivalently an unsuccessful search would abort the `Delete` operation). Thus m below should be considered to be $m = b$ for an (a, b) -tree, $m = 4$ for a $(2, 3, 4)$ -tree and $m = B$ for a B -tree. We avoid using a prefix such as `BST` or `AVL` to describe the relevant operation. Let T be an instance of a search tree of bounded depth (and also height). We deviate from the discussion of `BSTDelete(z)` or `AVLDelete(z)` as follows.

A `Delete(k, r)` operation initiates a `Search(k, r)` operation in T . If the `Search` terminates unsuccessfully, no key k was found, and `Delete` returns unsuccessfully. Otherwise key k has been located in T at node w . Contrary to `BSTDelete` that always utilizes the `Successor`, we may use in this scheme either the `Successor` or the `Predecessor`. Just like a `BSTInsert` is similar to an unsuccessful `BSTSearch`, for an `Insert` operation a `Search` is realized that eventually will prove unsuccessful. Thus connecting with `Search` we have reached an external node (a leaf) previously referred to as node c_* . We describe the operation in an agnostic way. Thus m below should be interpreted as an m for a MWST, a $m = b$ for an (a, b) -tree, a $m = 4$ for a $(2, 3, 4)$ -tree and $m = B$ for a B -tree.

Case 1: key k is at a node of the lowest level of T . If key k is the i -th item of w i.e. $k_{i-1} < k = k_i < k_{i+1}$, we check whether c_{i-1} and c_i point to external nodes (NULL, leaves). In such a case we directly delete the item with key k from w .

Case 2: keys k is NOT at node of the lowest level of T . Then, we act similar to the third case of `BSTDelete` except that we may use either the predecessor or the successor of k . We do this because the successor or the predecessor of k are stored at the bottomest node of the tree (just above the NULL leaves also referred to as external nodes.) In the remainder, v would be the node that contains the successor of k , and note that k is at node w . First we copy the key `succ(k)` into the item storing k at w . The item previously storing `succ(k)` becomes invalid and can be deleted/spliced out from v . Therefore `Delete(k, r)` is then reduced to `Delete(succ(k), v)` (or if we had used the predecessor instead, `Delete(pred(k), v)`).

Bounded depth search trees

Deletion operation

Continued. Whether it is Case 1 or Case 2, deleting a key value, can be resolved in one of two ways: (a) by a **transfer operation**, that transfers to say node w a key from the left or right sibling to replace the deleted k and make w has fewer keys than the minimum threshold allowed, or (b) by a **fusion operation**, that joins two nodes with few key values (below the threshold) into one that is still below or at the maximum threshold. The fusion operation removes a key value from the parent of the fused nodes. We might have a propagation to the parent of the node in question that eventually reaches the root of T and collapses the root thus reducing the height of the tree by one. ■

Parenthesis. Let us explain for an (a, b) -tree the restriction that $2 \leq a \leq (b + 1)/2$. If we have two sibling nodes that are a -nodes and thus each one has $a - 1$ keys values, and a keys is delete by the first of them then that node becomes an $(a - 1)$ -node and the other node remains an a -node. After the fusion into one node they have $(a - 1) + a = 2a - 1$ children. This is not an issue since $2a - 1 \leq b$. This is because $a \leq (b + 1)/2$. In terms of number of comparisons and other operations performed Delete is $O(h \times O(\lg m))$ which is $O(\lg n)$ just like the probing operations and Insert earlier.

DRAFT

$(2, 3, 4)$ -tree Delete

Example 7 ($(2, 3, 4)$ -tree Delete). The pseudocode for $(2, 3, 4)$ -tree Delete follows. (This is Delete with $m = 4$.)

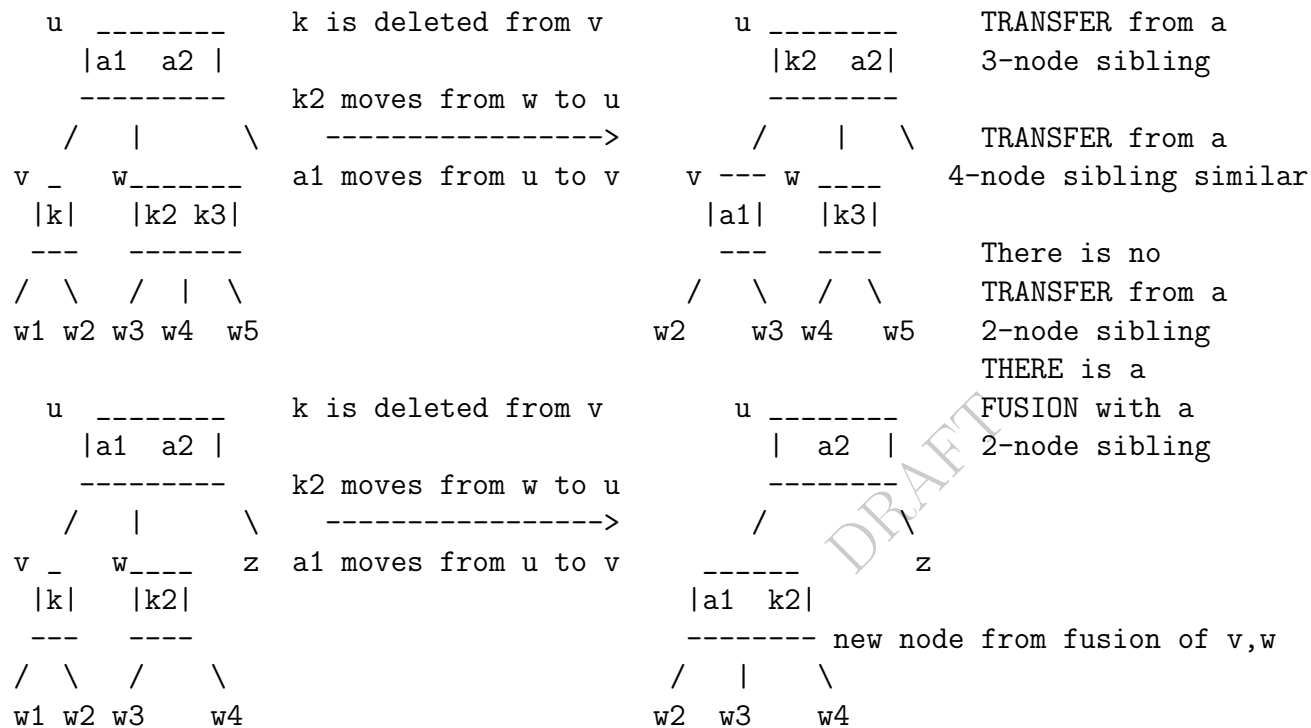
```
Proof. 2-3-4Delete(k, root(T))           // Delete (k,x) from T
1.  if (Search(k, root(T)) == NotFound)
2.      Do-Nothing-and-Return();
3.  else
4.      If node z storing k has internal nodes as children {
5.          Let (ki, xi) be the item ki=k
6.          Through the i-th child of z locate the rightmost internal node v
7.          whose children are external nodes.
8.          Determine (qv, xv) the last item of v.
9.          Swap (k, xi) of z with (qv, xv) of v
10.         RECORD that v contains (k, xi);
11.     }
12. else { // z has external nodes as children
13.     UPDATE v = z and make a note of (k, xi) // Rename v node containing k to z
13. }
14. Remove(k, v, T);

    Remove(k, v, T)
1.  Find in v (k, *), erase it, and also erase the corresponding external node.
2.  if v becomes an 1-node { // i.e. if v had one key and now it has zero and is empty!
3.      // We have an underflow condition: A transfer or a fusion can resolve it.
4.      Determine a sibling w of v that is a 3-node or a 4-node
5.      If such a sibling w exists
6.          Transfer(k, v, w);
7.      else
8.          Fusion(k, v); // During the fusion value a1 (see next page) is removed from parent u of v
9.          Remove(a1, u, T) // This call determines whether we need a Transfer or Fusion for u.
```

■

(2,3,4)-tree Delete case analysis

Proof continued.

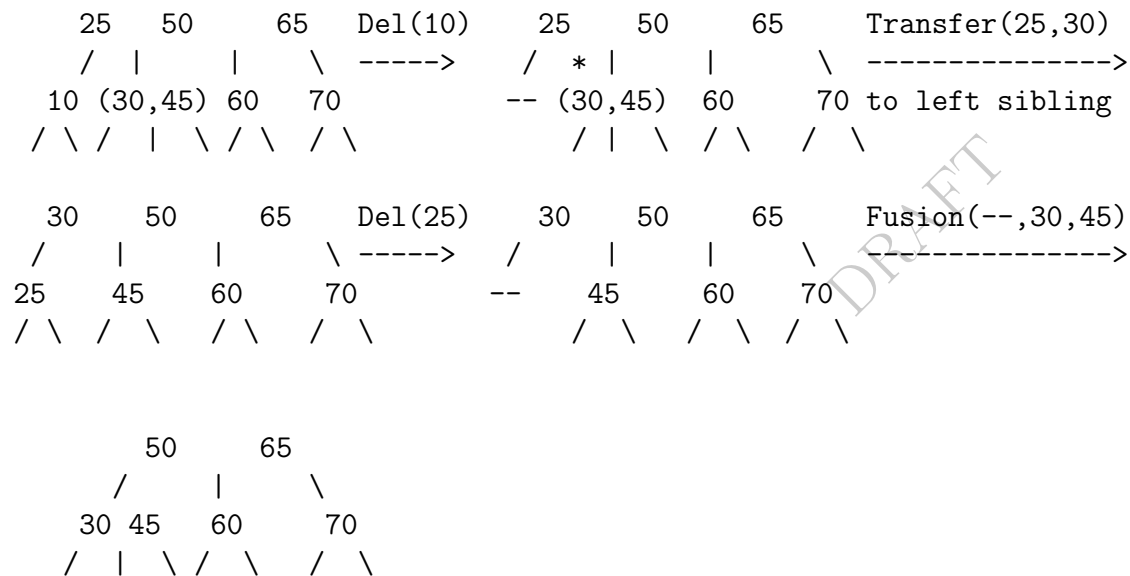


■

(2, 3, 4)-tree Delete example

Example 8. An example of Delete involving key with value 10 and then key with value 25 from a (2, 3, 4)-tree is shown. It continues the Insert example.

Proof.



■

Notice

Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

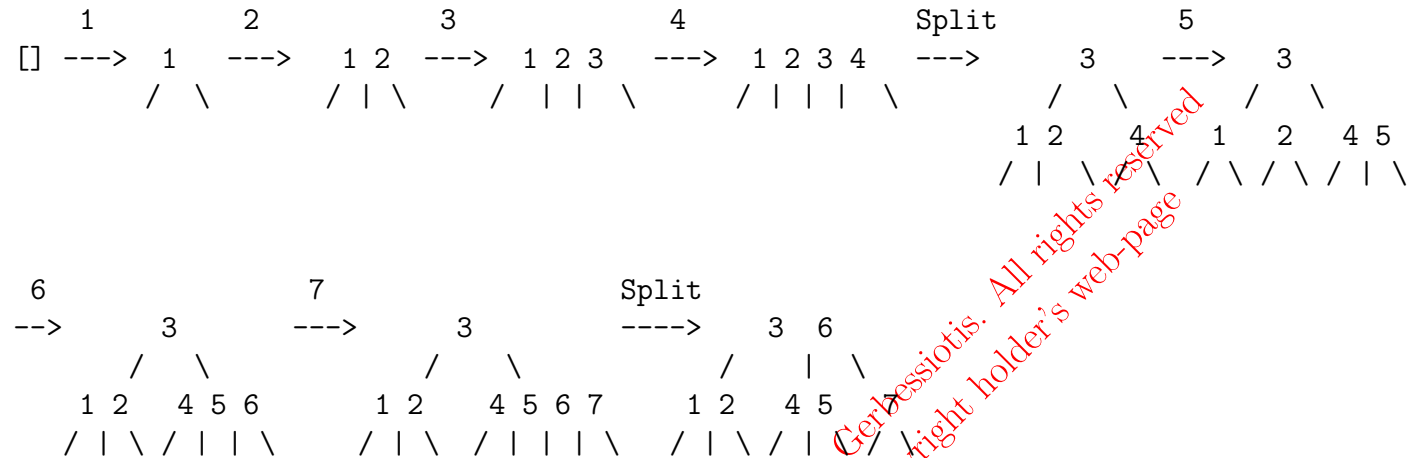
Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 1.

Insert 1, 2, 3, 4, 5, 6, 7. Show the steps. Then in Problem 2 delete 7.

Proof.

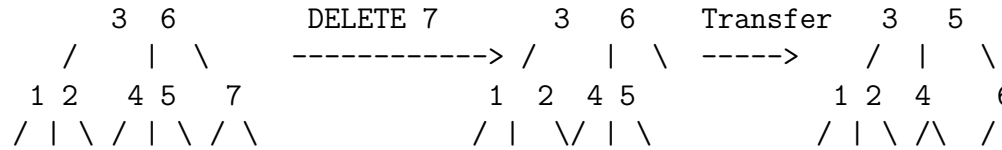


■

Problem Set
Problem 2.

Delete 7 from the tree of Problem 1.

Proof.



■

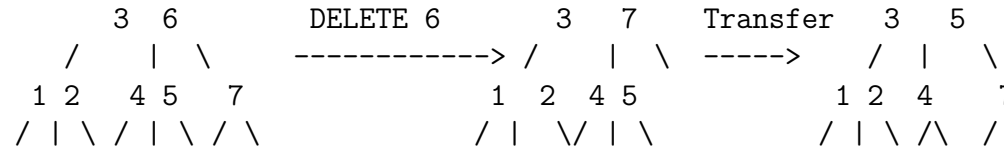
Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 3.

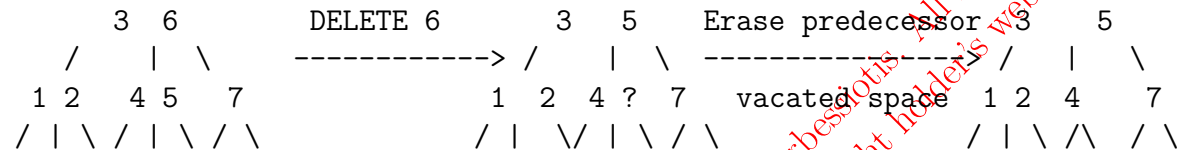
Delete 6 from the tree of Problem 1.

Proof. Either the successor or predecessor of z can be used!

Successor of 6 is used as 6 is not on the level above the NULLs /leaves



Predecessor of 6 is used



■