Generic AVL Tree Implementation (GAVL)

Pavel Pisa
pisa@cmp.felk.cvut.cz
Generic AVL Tree Implementation (GAVL)
by Pavel Pisa

Copyright © 2003 by Pavel Pisa

Implementation of AVL tree with some interesting features. All algorithms for insertion, deletion, traversal etc. are implemented without function call recursion or tree node stacks. The only exception is routine to check correctness of tree used for debugging purposes. The implementation has more advantages. It can work with partially unbalanced trees with balance factor (height difference) outside range <\(-1,1\)>. The balancing subroutine is common for insertion and deletion. Provided macros enables to build trees with strict item type checking which avoids necessity to use type-casting from (void *).
List of Figures

4-1. Tree balancing case 1 ................................................................. 29
4-2. Tree balancing case 2 ................................................................. 31

List of Equations

4-1. ........................................................................................................ 29
4-2. ........................................................................................................ 29
4-3. ........................................................................................................ 30
4-4. ........................................................................................................ 30
4-5. ........................................................................................................ 30
4-6. ........................................................................................................ 30
4-7. ........................................................................................................ 31
4-8. ........................................................................................................ 31
4-9. ........................................................................................................ 31
4-10. ...................................................................................................... 31
4-11. ...................................................................................................... 31
4-12. ...................................................................................................... 32
4-13. ...................................................................................................... 32
4-14. ...................................................................................................... 33
4-15. ...................................................................................................... 33
4-16. ...................................................................................................... 33
Chapter 1. Introduction

There are more concepts how to store data items sorted by some key values. The most commonly used ones are:

- sorted continuous arrays of items or pointers to items
- binary search trees (splash tree, AVL-tree, RB-tree)
- more childs per node search trees (B-tree)

The binary search trees are considered most efficient for in memory operations when frequent tree updates are present. If update frequency is very low, sorted arrays of pointers could be more efficient. If the node retrieval operations are time consuming the n-nary search trees could be used.

The provided GAVL library tries to offer fast and API friendly set of functions and macros usable for in-memory stored AVL trees of user specified items. The tree node chaining informations must be added to each stored item for holding of the tree node topology. The GAVL library can use user specified field of items or small malloced structure for holding of node chaining information.

The tree search, insert and delete functions can work with and enforce tree with unique items or can be used in mode when more items with same search key value are allowed. The balancing and other functions are written such way, that they tolerate even degraded tree which does not fulfill AVL tree definition. The operations do not result in further degradation or program exceptions. Even future tree operations tends to enhance balancing if the height differences (balance factors) of degraded tree has been correctly updated.
Chapter 2. Functions Description

Generic AVL tree

struct gavl_node

Name

struct gavl_node — Structure Representing Node of Generic AVL Tree

Synopsis

struct gavl_node {
    struct gavl_node * left;
    struct gavl_node * right;
    struct gavl_node * parent;
    int hdiff;
};

Members

left
    pointer to left child or NULL
right
    pointer to right child or NULL
parent
    pointer to parent node, NULL for root
hdiff
    difference of height between left and right child

Description

This structure represents one node in the tree and links left and right to nodes with lower and higher value of order criterion. Each tree is built from one type of items defined by user. User can decide to include node structure inside item representation or GAVL can malloc node structures for each inserted item. The GAVL allocates memory space with capacity
Chapter 2. Functions Description

sizeof(gavl_node_t)+sizeof(void*) in the second case. The item pointer is stored following node structure (void**)(node+1);

**struct gavl_root**

**Name**

`struct gavl_root` — Structure Representing Root of Generic AVL Tree

**Synopsis**

```c
struct gavl_root {
    gavl_node_t * root_node;
    int node_offs;
    int key_offs;
    gavl_cmp_fnc_t * cmp_fnc;
};
```

**Members**

- **root_node**
  
  pointer to root node of GAVL tree

- **node_offs**
  
  offset between start of user defined item representation and included GAVL node structure. If negative value is stored there, user item does not contain node structure and GAVL manages standalone ones with item pointers.

- **key_offs**
  
  offset to compared (ordered) fields in the item representation

- **cmp_fnc**
  
  function defining order of items by comparing fields at offset `key_offs`.
gavl_node2item

Name

gavl_node2item — Conversion from GAVL Tree Node to User Defined Item

Synopsis

void * gavl_node2item (const gavl_root_t * root, const gavl_node_t * node);

Arguments

root
GAVL tree root

node
node belonging to root GAVL tree

Return Value

pointer to item corresponding to node

gavl_node2item_safe

Name

gavl_node2item_safe — Conversion from GAVL Tree Node to User Defined Item

Synopsis

void * gavl_node2item_safe (const gavl_root_t * root, const gavl_node_t * node);
Arguments

\textit{root} \\
GAVL tree root

\textit{node} \\
node belonging to \textit{root} GAVL tree

Return Value

pointer to item corresponding to node

gavl\_node2key

Name

gavl\_node2key — Conversion from GAVL Tree Node to Ordering Key

Synopsis

\texttt{void * gavl\_node2key (const gavl\_root\_t * root, const gavl\_node\_t * node);}

Arguments

\textit{root} \\
GAVL tree root

\textit{node} \\
node belonging to \textit{root} GAVL tree

Return Value

pointer to key corresponding to node
gavl_next_node

Name

gavl_next_node — Returns Next Node of GAVL Tree

Synopsis

gavl_node_t * gavl_next_node (const gavl_node_t * node);

Arguments

node

node for which accessor is looked for

Return Value

pointer to next node of tree according to ordering

gavl_prev_node

Name


gavl_prev_node — Returns Previous Node of GAVL Tree

Synopsis


gavl_node_t * gavl_prev_node (const gavl_node_t * node);
Arguments

node

node for which predecessor is looked for

Return Value

pointer to previous node of tree according to ordering

\texttt{gavl\_balance\_one}

Name

\texttt{gavl\_balance\_one} — Balance One Node to Enhance Balance Factor

Synopsis

\begin{verbatim}
int gavl_balance_one (gavl_node_t ** subtree);
\end{verbatim}

Arguments

subtree

pointer to pointer to node for which balance is enhanced

Return Value

returns nonzero value if height of subtree is lowered by one
**gavl_insert_primitive_at**

**Name**

* gavl_insert_primitive_at — Low Level Routine to Insert Node into Tree

**Synopsis**

```c
int gavl_insert_primitive_at (gavl_node_t **root_nodep, gavl_node_t *node, gavl_node_t *where, int leftright);
```

**Arguments**

* root_nodep
  pointer to pointer to GAVL tree root node
* node
  pointer to inserted node
* where
  pointer to found parent node
* leftright
  left (>=1) or right (<=0) branch

**Description**

This function can be used for implementing AVL trees with custom root definition. The value of the selected *left* or *right* pointer of provided *node* has to be NULL before insert operation, i.e. node has to be end node in the selected direction.

**Return Value**

positive value informs about success
gavl_delete_primitive

Name

\texttt{gavl\_delete\_primitive} — Low Level Deletes/Unlinks Node from GAVL Tree

Synopsis

\begin{verbatim}
int gavl_delete_primitive (gavl_node_t ** root_nodep, gavl_node_t * node);
\end{verbatim}

Arguments

\begin{itemize}
  \item \texttt{root\_nodep}
    \begin{itemize}
      \item pointer to pointer to GAVL tree root node
    \end{itemize}
  \item \texttt{node}
    \begin{itemize}
      \item pointer to deleted node
    \end{itemize}
\end{itemize}

Return Value

positive value informs about success.

gavl_cut_first_primitive

Name

\texttt{gavl\_cut\_first\_primitive} — Low Level Routine to Cut First Node from Tree

Synopsis

\begin{verbatim}
gavl_node_t * gavl_cut_first_primitive (gavl_node_t ** root_nodep);
\end{verbatim}
Chapter 2. Functions Description

Arguments

root_nodep

pointer to pointer to GAVL tree root node

Description

This enables fast delete of the first node without tree balancing. The resulting tree is degraded but height differences are kept consistent. Use of this function can result in height of tree maximally one greater the tree managed by optimal AVL functions.

Return Value

returns the first node or NULL if the tree is empty

gavl_first_node

Name

gavl_first_node — Returns First Node of GAVL Tree

Synopsis


gavl_node_t * gavl_first_node (const gavl_root_t * root);

Arguments

root

GAVL tree root

Return Value

pointer to the first node of tree according to ordering
**gavl_last_node**

**Name**

gavl_last_node — Returns Last Node of GAVL Tree

**Synopsis**

```c
gavl_node_t * gavl_last_node (const gavl_root_t * root);
```

**Arguments**

`root`

GAVL tree root

**Return Value**

pointer to the last node of tree according to ordering

---

**gavl_is_empty**

**Name**

gavl_is_empty — Check for Empty GAVL Tree

**Synopsis**

```c
int gavl_is_empty (const gavl_root_t * root);
```
Arguments

*root*

GAVL tree root

Return Value

returns non-zero value if there is no node in the tree

**gavl_search_node**

Name

gavl_search_node — Search for Node or Place for Node by Key

Synopsis

```c
int gavl_search_node (const gavl_root_t * root, const void * key, int mode, gavl_node_t ** nodep);
```

Arguments

*root*

GAVL tree root

*key*

key value searched for

*mode*

mode of the search operation

*nodep*

pointer to place for storing of pointer to found node or pointer to node which should be parent of inserted node
Description

Core search routine for GAVL trees searches in tree starting at root for node of item with value of item field at offset key_off equal to provided *key value. Values are compared by function pointed by *cmp_fnc field in the tree root. Integer mode modifies search algorithm: GAVL_FANY .. finds node of any item with field value *key, GAVL_FFIRST .. finds node of first item with *key, GAVL_FAFTER .. node points after last item with *key value, reworded - index points at first item with higher value of field or after last item

Return Value

Return of nonzero value indicates match found. If the mode is ored with GAVL_FCMP, result of last compare is returned.

gavl_find

Name

gavl_find — Find Item for Provided Key

Synopsis

void * gavl_find (const gavl_root_t * root, const void * key);

Arguments

root

GAVL tree root

key

key value searched for

Return Value

pointer to item associated to key value.
gavl_find_first

Name

*gavl_find_first* — Find the First Item with Provided Key Value

Synopsis

```c
void * gavl_find_first (const gavl_root_t * root, const void * key);
```

Arguments

- **root**
  - GAVL tree root
- **key**
  - key value searched for

Description

same as above, but first matching item is found.

Return Value

pointer to the first item associated to key value.

gavl_find_after

Name

*gavl_find_after* — Find the First Item with Higher Key Value
Chapter 2. Functions Description

Synopsis

void * gavl_find_after (const gavl_root_t * root, const void * key);

Arguments

root
GAVL tree root

key
key value searched for

Description

same as above, but points to item with first key value above searched key.

Return Value

pointer to the first item associated to key value.

gavl_insert_node_at

Name

gavl_insert_node_at — Insert Existing Node to Already Computed Place into GAVL Tree

Synopsis

int gavl_insert_node_at (gavl_root_t * root, gavl_node_t * node, gavl_node_t * where, int leftright);
Arguments

root
GAVL tree root

node
pointer to inserted node

where
pointer to found parent node

lefright
left (1) or right (0) branch

Return Value
positive value informs about success

gavl_insert_node

Name
gavl_insert_node — Insert Existing Node into GAVL Tree

Synopsis

int gavl_insert_node (gavl_root_t * root, gavl_node_t * node, int mode);

Arguments

root
GAVL tree root

node
pointer to inserted node
Chapter 2. Functions Description

mode

if mode is GAVL_FAFTER, multiple items with same key can be used, else strict ordering is required

Return Value

positive value informs about success

gavl_insert

Name

gavl_insert — Insert New Item into GAVL Tree

Synopsis

int gavl_insert (gavl_root_t * root, void * item, int mode);

Arguments

root

GAVL tree root

item

pointer to inserted item

mode

if mode is GAVL_FAFTER, multiple items with same key can be used, else strict ordering is required

Return Value

positive value informs about success, negative value indicates malloc fail or attempt to insert item with already defined key.
gavl_delete_node

Name

gavl_delete_node — Deletes/Unlinks Node from GAVL Tree

Synopsis

int gavl_delete_node (gavl_root_t * root, gavl_node_t * node);

Arguments

root
GAVL tree root

node
pointer to deleted node

Return Value

positive value informs about success.

gavl_delete

Name

gavl_delete — Delete/Unlink Item from GAVL Tree
Synopsis

\texttt{int gavl\_delete (gavl\_root\_t * root, void * item);} 

Arguments

\textit{root}  
GAVL tree root

\textit{item}  
pointer to deleted item

Return Value

positive value informs about success, negative value indicates that item is not found in tree defined by root

\texttt{gavl\_delete\_and\_next\_node}

Name

\texttt{gavl\_delete\_and\_next\_node} — Delete/Unlink Item from GAVL Tree

Synopsis

\texttt{gavl\_node\_t * gavl\_delete\_and\_next\_node (gavl\_root\_t * root, gavl\_node\_t * node);} 

Arguments

\textit{root}  
GAVL tree root
node

pointer to actual node which is unlinked from tree after function call, it can be unallocated or reused by application code after this call.

Description

This function can be used after call gavl_first_node for destructive traversal through the tree, it cannot be combined with gavl_next_node or gavl_prev_node and root is emptied after the end of traversal. If the tree is used after unsuccessful/unfinished traversal, it must be balanced again. The height differences are inconsistent in other case. If traversal could be interrupted, the function gavl_cut_first could be better choice.

Return Value

pointer to next node or NULL, when all nodes are deleted

gavl_cut_first

Name
gavl_cut_first — Cut First Item from Tree

Synopsis

void * gavl_cut_first (gavl_root_t * root);

Arguments

root

GAVL tree root
Chapter 2. Functions Description

Description

This enables fast delete of the first item without tree balancing. The resulting tree is degraded but height differences are kept consistent. Use of this function can result in height of tree maximally one greater the tree managed by optimal AVL functions.

Return Value

returns the first item or NULL if the tree is empty

Custom AVL Tree Instances

The provided macros allows to define all AVL tree functions for tree with user defined root and item types. The next defined function names are prefixed by custom selected cust_prefix:

```c
cust_item_t * cust_prefix_node2item(const cust_root_t * root, const gavl_node_t * node);
```

Conversion from AVL tree node to user custom defined item

```c
cust_key_t * cust_prefix_node2key (const cust_root_t * root, const gavl_node_t * node);
```

Conversion from AVL tree node to pointer to defined ordering key

```c
int cust_prefix_search_node (const cust_root_t * root, const cust_item_t * key, gavl_node_t ** nodep);
```

Search for node or place for node by key

```c
cust_item_t * cust_prefix_find (const cust_root_t * root, const cust_key_t * key);
```

Pointer to item with key value or NULL
cust_item_t * cust_prefix_find_first (const cust_root_t * root, const cust_key_t * key);

Same as above, but first matching item is found.

cust_item_t * cust_prefix_find_after (const cust_root_t * root, const cust_key_t * key);

Finds the first item with key value higher than value pointed by key or returns NULL.

int cust_prefix_insert (cust_root_t * root, cust_item_t * item);

Insert new item into AVL tree. If the operation is successful positive value is returned. If key with same key value already exists negative value is returned

int cust_prefix_delete_node (cust_root_t * root, gavl_node_t * node);

Deletes/unlinks node from custom AVL tree

int cust_prefix_delete (cust_root_t * root, cust_item_t * item);

Deletes/unlinks item from custom AVL tree

gavl_node_t * cust_prefix_first_node (const cust_root_t * root);

Returns the first node or NULL if the tree is empty

gavl_node_t * cust_prefix_last_node (const cust_root_t * root);

Returns last node or NULL if the tree is empty

cust_item_t * cust_prefix_first(const cust_root_t * root);

Returns pointer to the first item of the tree or NULL
cust_item_t * cust_prefix_last(const cust_root_t * root);

Returns pointer to last item of the tree or NULL

cust_item_t * cust_prefix_next(const cust_root_t * root, const cust_item_t * item);

Returns pointer to next item of the tree or NULL if there is no more items

cust_item_t * cust_prefix_prev(const cust_root_t * root, const cust_item_t * item);

Returns pointer to previous item of the tree or NULL if there is no more items

int cust_prefix_is_empty (const cust_root_t * root);

Returns non-zero value if there is no node in the tree

cust_item_t * cust_prefix_cut_first (cust_root_t * root);

Returns the first item or NULL if the tree is empty. The returned item is unlinked from the tree without tree rebalance

GAVL_CUST_NODE_INT_IMP

Name

GAVL_CUST_NODE_INT_IMP — Implementation of new custom tree with internal node

Synopsis

GAVL_CUST_NODE_INT_IMP ( cust_prefix, cust_root_t, cust_item_t, cust_key_t, cust_root_node, cust_item_node, cust_item_key, cust_cmp_fnc);
Arguments

cust_prefix
    defines prefix for builded function names

cust_root_t
    user defined structure type of root of the tree

cust_item_t
    user defined structure type of items stored in the tree

cust_key_t
    type of the key used for sorting of the items

cust_root_node
    the field of the root structure pointing to the tree root node

cust_item_node
    the field of item structure used for chaining of items

cust_item_key
    the key field of item structure defining order of items

cust_cmp_fnc
    the keys compare function

Description

There are two macros designed for building custom AVL trees. The macro GAVL_CUST_NODE_INT_DEC declares functions for custom tree manipulations and is intended for use in header files. The macro GAVL_CUST_NODE_INT_IMP builds implementations for non-inlined functions declared by GAVL_CUST_NODE_INT_DEC. The cust_cmp_fnc is used for comparison of item keys in the search and insert functions. The types of two input arguments of cust_cmp_fnc functions must correspond with cust_key_t type. Return value should be positive for case when the first pointed key value is greater then second, negative for reverse case and zero for equal pointed values.
Chapter 3. Examples of GAVL Usage

Generic AVL Tree

This chapter describes use of generic AVL tree. This tree has advantage, that same functions could operate with any user provided items, but operations return type is void* and needs type-casting to user item type.

```c
#include <malloc.h>
#include "ul_gavl.h"
```

The above code fragment includes basic GAVL declarations.

```c
typedef struct test1_item {
   int my_val;
   /* more user data ... */
} test1_item_t;
```

New item type is declared. Type have to contain or be connected to some key value. The integer number my_val will be used as key value in this example. The tree root instance definition follows.

```c
gavl_root_t test1_tree={
   .root_node = NULL,
   .node_offs = -1,
   .key_offs = UL_OFFSETOF(test1_item_t, my_val),
   .cmp_fnc = gavl_cmp_int
};
```

The field root_node have to be initialized to NULL. The value -1 for the node_offs field directs GAVL functions to allocate external node structure for each inserted item. The macro UL_OFFSETOF computes offset of my_val field in test1_item_t item. The last assignment select one of predefined functions for key values comparison.

Almost same declarations and definitions could be used if the node chaining structure is contained inside item structure. That is shown in the next code fragments.

```c
typedef struct test2_item {
   gavl_node_t my_node;
   int my_val;
   /* more user data ... */
} test2_item_t;
```

The item declaration contains field with gavl_node_t type in this case. This field is used for storage of tree topology information.

```c
gavl_root_t test2_tree={
   .root_node = NULL,
   .node_offs = UL_OFFSETOF(test2_item_t, my_node),
   .key_offs = UL_OFFSETOF(test2_item_t, my_val),
   .cmp_fnc = gavl_cmp_int
};
```
The field node_offs contains offset of the node structure inside item now. Next fragments are part of the function working with defined tree. There would be no difference in the functions calls for items with external nodes.

cust2_item_t *item;
int i;
gavl_node_t *node;

Declare some local variables first. Insert 100 items into tree then.

for(i=0;i<items_cnt;i++){
    item=malloc(sizeof(test2_item_t));
    item->my_val=i;
    if(gavl_insert(&test2_tree,item,0)<0)
        printf("gavl_insert error\n");
}

The tree is expected to store items with unique key values. If more items with same key values should be allowed, then the value of mode parameter in gavl_insert function call have to be changed from 0 to GAVL_PAFTER.

for(i=0;i<102;i++){
    if(!item=(cust2_item_t *)gavl_find(&test2_tree,&i))
        printf("no item \%d\n",i);
    else
        if(item->my_val!=i)
            printf("gavl_find mismatch\n");
}

Some test of retrieving item corresponding to specified key. The tree in order traversal is in the next code sample.

node=gavl_first_node(&test2_tree)
while(node){
    item=(cust2_item_t *)gavl_node2item(&test2_tree,node);
    /* do something with item
     * insert and delete allowed there except delete of the actual item
     */
    node=gavl_next_node(node);
    /* do something with item
     * insert and delete allowed there except delete of next item
     */
}

Example of the item deletion is in the next fragment. The function gavl_delete guarantees that trial of deleting item, which is not part of the tree, is detected and negative value is returned. There is one prerequisite for this behavior for items with internal nodes that have never been part of any tree. The field parent of node structure stored in the item have to be initialized to NULL value.

for(i=0;i<80;i++){
Chapter 3. Examples of GAVL Usage

```c
if(!(item=(cust2_item_t *)gavl_find(&test2_tree,&i)))
    continue;
if(gavl_delete(&test2_tree,item)<0)
    printf("gavl_delete error\n");
free(item);
```

The function `gavl_cut_first` can significantly simplify deletion of all nodes of the tree without need of traversal over the tree. This function does not rebalance the tree, but keeps tree consistent.

```c
while((item=(cust2_item_t *)gavl_cut_first(&test2_tree))!=NULL)
    free(item);
```

## Customized AVL Tree

The function templates are provided for faster and type-safe custom tree definition and manipulation. These templates expect user defined items with internal node structures and key comparison function with key pointer input parameters. The type of these parameters is required to be pointer to the key type.

Next code fragment is intended to be part of the declarations of user defined custom tree and items for some data storage purpose. The header with these definitions should be included from all modules directly working with specified tree.

```c
#include "ul_gavl.h"

typedef int cust_key_t;

typedef struct cust2_item {
    cust2_key_t my_val;
    gavl_node_t my_node;
    /* more user item data ... */
} cust2_item_t;

typedef struct cust2_root {
    gavl_node_t *my_root;
    /* more user root/tree data ... */
} cust2_root_t;

int cust_cmp_fnc(cust_key_t *a, cust_key_t *b);
```

As can be seen from above code fragment, the key field with user declared type (cust_key_t) and internal node structure are required for items. The only one field with type gavl_node_t* is required for the custom tree root structure.

The use of the macro `GAVL_CUST_NODE_INT_DEC` declares all tree manipulation functions for the custom tree. The function names are prefixed by prefix specified as the first parameter of macro invocation. This declaration can be included in user header files as well.

```c
GAVL_CUST_NODE_INT_DEC(cust2, cust2_root_t, cust2_item_t, cust2_key_t,\
    my_root, my_node, my_val, cust_cmp_fnc)
```
Chapter 3. Examples of GAVL Usage

The implementation of the custom tree functions is realized by use of macro GAVL_CUST_NODE_INT_IMP with same parameters as for GAVL_CUST_NODE_INT_DEC macro.

```c
#include "ul_gavlcust.h"

GAVL_CUST_NODE_INT_IMP(cust2, cust2_root_t, cust2_item_t, cust2_key_t, my_root, my_node, my_val, cust_cmp_fnc)

cust2_root_t cust2_tree;
```

Static tree root instance is included at last line of above code fragment as well.

The next sample function for the custom tree shows same operation as have been described for generic tree.

```c
void test_cust_tree(void)
{
    int i;
    cust2_item_t *item;

    /* build tree */
    for(i=1;i<=100;i++)
    {
        item=malloc(sizeof(cust2_item_t));
        item->my_val=i;
        if(cust2_insert(&cust2_tree,item)<0)
            printf("cust2_insert error\n");
    }

    /* traverse tree */
    printf("Custom tree cust2:\n");
    for(item=cust2_first(&cust2_tree);item;item=cust2_next(&cust2_tree,item))
        printf("%d ",item->my_val);
    printf("\n");

    /* delete part of the tree */
    for(i=1;i<=80;i++)
    {
        item=cust2_find(&cust2_tree,&i);
        if(cust2_delete(&cust2_tree,item)<0)
            printf("cust2_delete error\n");
        free(item);
    }

    /* traverse in reverse order */
    printf("Custom tree cust2:\n");
    for(item=cust2_last(&cust2_tree);item;item=cust2_prev(&cust2_tree,item))
        printf("%d ",item->my_val);
    printf("\n");

    /* delete all remaining items */
    while((item=cust2_cut_first(&cust2_tree))!=NULL)
        free(item);
}
```
Chapter 4. Used Algorithms Background

Tree Balancing Operations

This section describes used algorithms for keeping AVL tree balanced.

Next convention is used for height and balance computations for node X. The height of subtree starting at X is labeled as \( h(X) \). Height difference (balance factor) for node X is labeled as \( \Delta = h(left(X)) - h(right(X)) \).

Tree Balancing Case 1

Figure 4-1. Tree balancing case 1

Before operation

After operation

\[
\Delta_s = n - b \geq 2 \\
\Delta_n = a - p \geq 0
\]

The height of subtree S is defined by highest branch, which is branch growing from node A. This leads to next equations.

\[
n = a + 1 \\
s = a + 2
\]
\[ p = a - \Delta n \]
\[ b = a + 1 - \Delta s \]

The height of branches N and S is marked as \( n_1 \) and \( s_1 \) after balancing.

\[ \Delta s_1 = p - b \]
\[ \Delta n_1 = a - s_1 \]

\[ s_1 = \max(p, b) + 1 \]
\[ s_1 = \max(a - \Delta n, a + 1 - \Delta s) + 1 \]
\[ s_1 = a + 1 + \max(-\Delta n, 1 - \Delta s) \]
\[ s_1 = a + 1 - \min(\Delta n, \Delta s - 1) \]

\[ \Delta s_1 = a - \Delta n - a - 1 + \Delta s \]
\[ \Delta s_1 = \Delta s - \Delta n - 1 \]
\[ \Delta n_1 = a - a - 1 + \min(\Delta n, \Delta s - 1) \]
\[ \Delta n_1 = \min(\Delta n - 1, \Delta s - 2) \]
\[ \Delta n_1 = \begin{cases} \Delta n - 1 & \Delta s \geq \Delta n + 1 \\ \Delta s - 2 & \Delta s \leq \Delta n + 1 \end{cases} \]

Because balancing in case 1 does not guarantee that new tree has lower height than original tree, it is necessary to compute tree height change (tree height lowering).

\[ s - n_1 = s - (\max(a, s_1) + 1) \]
\[ = a + 2 - (\max(a, a + 1 + \max(-\Delta n, 1 - \Delta s)) + 1) \]
\[ = 1 - \max(0, \max(1 - \Delta n, 2 - \Delta s)) \]
\[ = \min(1, \Delta n, \Delta s - 1) \quad \Delta n \geq 0, \Delta s \geq 2 \]
\[ = \min(1, \Delta n) \]
\[ s - n_1 = \begin{cases} 1 & \Delta n \neq 0 \\ 0 & \Delta n = 1 \end{cases} \]
Tree Balancing Case 2

This case has advantage, that it reduces tree height for all node height combinations. Tree height is given by branch P B or P C.

\[ \Delta s = n - d \geq 2 \]
\[ \Delta n = a - p \leq -1 \]
\[ n = p + 1 \]
\[ s = p + 2 \]
\[ d = p + 1 - \Delta s \]
\[ a = p + \Delta n \]

For \( b \geq c \) \quad c = b - \Delta p \quad p = b + 1

For \( c \geq b \) \quad b = c + \Delta p \quad p = c + 1

When \( b \geq c \) then the height differences \( \Delta s1 \), \( \Delta n1 \) and \( \Delta p1 \) for nodes S, N and P of the balanced tree can be computed from next equations.
$b \geq c$, $\Delta p \geq 0$

\begin{align*}
n_1 &= b + 1 \\
p_1 &= b + 2 \\
s_1 &= \max(c, d) + 1 = \max(b - \Delta p, b + 2 - \Delta s) + 1 \\
\Delta s_1 &= c - d = b - \Delta p - b - 2 + \Delta s \\
\Delta s_1 &= \Delta s - 2 - \Delta p \\
\Delta n_1 &= a - b = b + 1 + \Delta n - b \\
\Delta n_1 &= \Delta n + 1 \\
\Delta p_1 &= n_1 - s_1 = b + 1 - \max(b - \Delta p, b + 2 - \Delta s) - 1 \\
\Delta p_1 &= \min(\Delta p, \Delta s - 2) \\
\Delta p_1 &= \Delta p \geq 0, \Delta s \geq 2
\end{align*}

When $c \geq b$ then the height differences $\Delta s_1$, $\Delta n_1$ and $\Delta p_1$ for nodes $S$, $N$ and $P$ of the balanced tree can be computed from next equations.

\begin{align*}
c \geq b$, $\Delta p \geq 0$

\begin{align*}
s_1 &= c + 1 \\
p_1 &= c + 2 \\
n_1 &= \max(a, b) + 1 = \max(c + 1 + \Delta n, c + \Delta p) + 1 \\
\Delta s_1 &= c - d = c - c - 2 + \Delta s \\
\Delta s_1 &= \Delta s - 2 \\
\Delta n_1 &= a - b = c + 1 + \Delta n - c - \Delta p \\
\Delta n_1 &= \Delta n + 1 - \Delta p \\
\Delta p_1 &= n_1 - s_1 = \max(c + 1 + \Delta n, c + \Delta p) + 1 - c - 1 \\
\Delta p_1 &= \max(\Delta n, \Delta s - 1) \\
\Delta p_1 &= \Delta n \leq -1, \Delta p \leq 0
\end{align*}

\textbf{Height of the AVL Tree}

Minimal number of nodes $T$ can be computed recursively.

\[ T(h) = T(h - 1) + T(h - 2) + 1, \quad T(0) = 0, \quad T(1) = 1 \]

1, 2, 4, 7, 12, 20, 33, 54, 88, 143, 232, 376, 609, 986
\[ T(h) = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^{(h+2)} - 1 \]

\[ T(h) = \frac{1}{\sqrt{5}} 2^{(h+2) \log_2 \left( \frac{1 + \sqrt{5}}{2} \right)} - 1 \]

\[ h = \left( \log_2(T + 1) - \log_2 \left( \frac{1}{\sqrt{5}} \right) \right) \log^{-1}_2 \left( \frac{1 + \sqrt{5}}{2} \right) - 2 \]

\[ \log^{-1}_2 \left( \frac{1 + \sqrt{5}}{2} \right) = 1.44042 \]

An AVL tree with \( n \) nodes has height between \( \log_2 (n + 1) \) and \( 1.44 \times \log_2 (n + 2) - 0.328 \). An AVL tree with height \( h \) has between \( 2^{(h + .328) / 1.44} - 1 \) and \( 2^h - 1 \) nodes.

A red-black tree with \( n \) nodes has height at least \( \log_2 (n + 1) \) but no more than \( 2 \times \log_2 (n + 1) \). A red-black tree with height \( h \) has at least \( 2^{h / 2} - 1 \) nodes but no more than \( 2^h - 1 \) nodes.