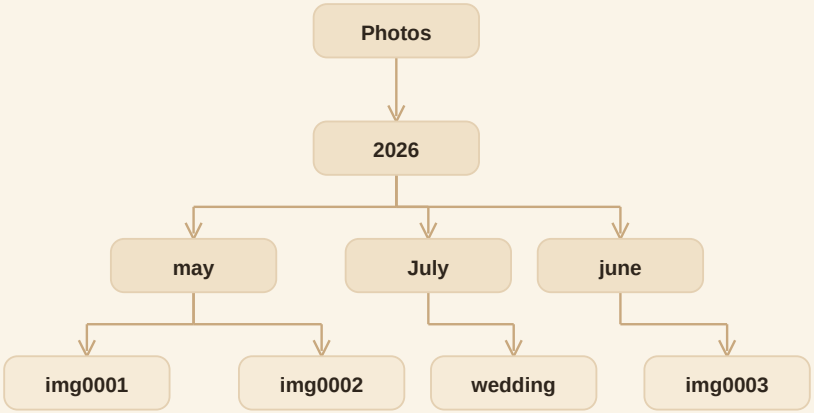
AUTOIT 3.3.16.1 - STRUCTURE OF ARRAYS - NO OBJECTS, NO DLL

Tree.au3

Quick start

version 0.2.0 | 32 public functions | 5 162 assertions passed | MIT licence

A tree of things you can already name: folders, employees, comments, menu items, file paths. It keeps every link in plain integer arrays, so a read costs about one microsecond, and it still gives you a name search that is not a scan. This booklet is seven short chapters: the nine words, one runnable program, the whole surface, and what to do when something goes wrong.



WHAT IS INSIDE

01 The nine words and one drawing, so the rest has somewhere to land

02 Ten minutes: put the file down, run the twelve-line program

03 Getting around: names, parents, children, paths, the whole tree

04 Growing and reshaping: add, move, remove, and the index rule

05 Why deleting a named node stays fast, with the measurements

06 Every error code, and eight things that look like bugs

07 The full function list, the benchmark, the files, how to rebuild this

NUMERIC1

https://www.autoitscript.com/forum/profile/118605-numeric1/

typeset with PDFX 1.4.0

Contents

01 One picture, nine words 03

02 Ten minutes, one file 05

03 Getting around 06

04 Growing and reshaping 09

05 Names, and why deleting stays fast 11

06 When it goes wrong 13

07 The whole surface, and the proof 15

How to read these ten pages

1. **Read chapter 01 once, it is one drawing.** Nothing else depends on it.
2. **Copy the twelve-line program of chapter 02 and run it.** If it prints four lines, the UDF works on your machine.
3. **Work through chapter 04 with your own data in front of you.** That is the chapter where a tree becomes useful.
4. **Keep chapter 06 open when something returns 0.** Every error number is in one table.
5. **Skip chapter 05 the first time.** It explains a design decision, not a function you must call.

> **TWO FILES ARE ENOUGH**
> Tree.au3 next to your script is the whole UDF. The booklet was built from the same folder that holds Tree_SelfTest.au3, Tree_Bench.au3 and Tree_CostProbe.au3; you never need those to use the tree, only to check the claims in chapter 07.

01

VOCABULARY

One picture, nine words

Everything in this UDF answers one of nine questions about a node. Read the drawing, read the table, and the function names become obvious.

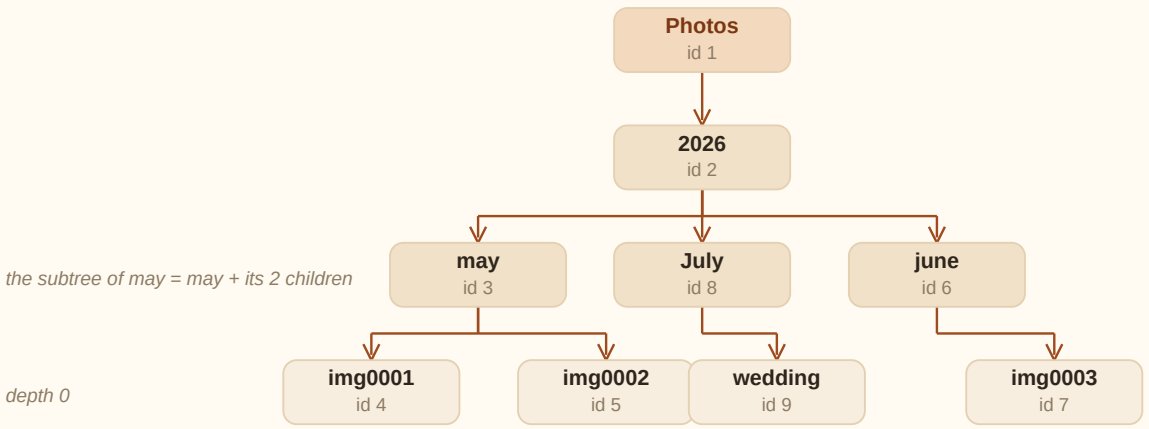


Figure 1. *The sample tree of chapter 02, with the ids the UDF handed out. ids are not assigned in reading order: 2026 has children may (id 3) and june (id 6) because img0001 and img0002 took ids 4 and 5 in between. Children are kept in the order you added them, and that is the only order the UDF promises.*

The nine words

WORD	WHAT IT MEANS HERE	THE CALL THAT ANSWERS IT
node	One thing in the tree: a folder, a person, a comment. It exists even if it holds nothing.	you hand out its id
id	A whole number. That is all a node is to the UDF: an index into arrays. 1 is the root.	Tree_Add returns one
root	The single node everything hangs from. Tree_Create makes it, and resets the whole tree.	Tree_Root()
parent	The node a node hangs from. Exactly one, never itself. The root has none.	Tree_Parent($i)
child	A node hanging directly under this one. In insertion order, always.	Tree_Children($i, $a)
sibling	Nodes that share one parent, visited left to right.	Tree_FirstChild / Tree_NextSibling
subtree	A node and everything below it, itself a tree.	Tree_SubtreeSize($i)
depth	How many parents to climb before the root. The root is 0.	Tree_Depth($i)
name	A label you choose, unique in the tree, so you can ask for a node by text.	Tree_Find("may")

What a node holds besides its links

Each node carries one data slot. Put anything AutoIt can hold in it - a string, a number, an array, even a Map - the UDF never reads it, it only hands it back. The name slot is a string and is the only part that is indexed.

Three things this UDF is not

- **It is not a tree view.** No drawing, no GUI item, no control. It holds the shape; you render it however you like.
- **It is not sorted.** Children come back in the order you added them. Sorting is yours, on the array Tree_Children fills.
- **It is not a database.** Nothing survives the script: use Tree_Export and Tree_Import, which are text files, not a store.

> **ONE TREE PER SCRIPT**
>
> The state is global on purpose: an accessor in AutoIt costs 10 to 15 microseconds, and giving every instance its own object would put that price back on every call. If you truly need two trees at once, run two scripts, or finish with one before creating the next - Tree_Create resets what is live.

02

TEN MINUTES

Ten minutes, one file

The goal of this chapter is one line of console output that proves the UDF works where you are sitting. Nothing else is asked of you.

Put the file down

1. **Copy Tree.au3 next to your own script.** Same folder, no subfolder, no configuration.
2. **At the top of your script write #include "Tree.au3"** AutoIt finds it there, then falls back to the Include folder.
3. **Call Tree_Create() once, before any other Tree_ call.** Without it every other call answers error 8.
4. **Run it with AutoIt3.exe your_script.au3.** Not /run: that switch does not exist and AutoIt exits with 1 and no message.

The twelve-line program

Type this, or open Tools\Tree_Starter.au3, which is exactly it. Comments are English, names are English, and there is no callback, no object and no DLL.

```
Tree_Create("Team")
Local $dev = Tree_Add(Tree_Root(), "development", "dev")
Tree_Add($dev, "Alice", "alice")
Tree_Add($dev, "Bob", "bob")
Local $boss = Tree_Find("dev")
ConsoleWrite("I am " & Tree_Name($boss) & ", my parent is " & Tree_Name(Tree_Parent($boss)) & @CRLF)
ConsoleWrite("dev has " & Tree_NChildren($dev) & " children and the whole tree has " & Tree_Count() & " nodes" & @CRLF)
```

Tree_Starter.au3, between its DOC markers. Verbatim, nothing trimmed.

What it prints here

ACTUAL OUTPUT, AUTOIT 3.3.16.1 X86, WINDOWS 10.0.19045

```
I am dev, my parent is Team
dev has 2 children and the whole tree has 4 nodes
```

Four nodes: the root Team, dev under it, and Alice and Bob under dev. Tree_Name($boss) says dev because Tree_Find returned the same node id that Tree_Add handed back. If you see those two lines, stop here and go to chapter 04: you already have a working install.

IF NOTHING PRINTS

A console window that stays empty usually means the script was started by double-clicking it, which detaches standard output. Start it from a terminal with AutoIt3.exe Tree_Starter.au3, or replace ConsoleWrite with MsgBox for one run. A "unknown function name" from Au3Check means Tree.au3 is not next to the script.

NEXT CHAPTER 03 Getting around

03

GETTING AROUND

Getting around

Sixteen calls, all of them read-only, none of them changing the tree. This is the chapter you will keep open.

One node in, one value out

CALL	ANSWERS	WHEN IT FAILS
Tree_Root()	id of the root, always 1	0 with error 8 if no tree yet
Tree_Count()	how many nodes are alive	never
Tree_Capacity()	how many nodes fit before the arrays double	never
Tree_Name($i)	its name, or an empty string	0 with error 1
Tree_GetData($i)	whatever you stored in it	error 1 only for a bad id
Tree_Parent($i)	the id it hangs from, 0 for the root	0 with error 1
Tree_Depth($i)	how many parents to climb	0 with error 1
Tree_NChildren($i)	how many nodes hang directly under it	0 with error 1
Tree_IsAncestor($a, $i)	1 when $a is above $i, 0 otherwise	error 1 for either bad id

By name, by path, by walk

CALL	ANSWERS	NOTE
Tree_Find("may")	the id of that name, or 0	a Map read, not a scan
Tree_Children($i, $aOut)	count, and fills $aOut[1..n] with ids	1-based, resized for you
Tree_PathTo($i, $aOut)	count, and fills the whole chain root first	same array convention
Tree_FirstChild / Tree_NextSibling	walk the children without an array	cheapest loop, 0 ends it
Tree_Walk("Func", $i)	calls your function on every node of that subtree	preorder; return less than 0 to stop
Tree_SubtreeSize($i)	how many nodes are under it, counting it	see chapter 04 for the cost

Finding one node, then its whole path

```
Local $hit = Tree_Find("img0002")
$s = "Find(img0002) -> id " & $hit
$s &= ", parent " & Tree_Name(Tree_Parent($hit))
$s &= ", data " & Tree_GetData($hit)
ConsoleWrite($s & @CRLF)

Local $aPath[1]
Local $n = Tree_PathTo($hit, $aPath)
Local $sPath = ""
For $k = 1 To $n
    If $k > 1 Then $sPath &= " > "
    $sPath &= Tree_Name($aPath[$k])
Next
ConsoleWrite("path from the root (" & $n & " steps): " & $sPath & @CRLF)
ConsoleWrite("may is an ancestor of img0002? " & Tree_IsAncestor($may, $hit) & @CRLF)
```

THE THREE LINES THIS BLOCK PRINTED

```
Find(img0002) -> id 5, parent may, data 2.0 MB
path from the root (4 steps): Photos > 2026 > may > img0002
may is an ancestor of img0002? 1
```

Tree_PathTo fills an array from the root down, which is why the first element is the root and the last is the node you asked about. Tree_IsAncestor does the same climb but stops as soon as it matches, so use it for a yes-or-no question instead of counting a path.

Printing the whole tree

```
ConsoleWrite("the whole tree, preorder:" & @CRLF)
Tree_Walk("_PrintNode")
    ; and the callback itself, a top-level function:
Func _PrintNode($iNode)
    Local $sPad = "", $k, $sData = Tree_GetData($iNode)
    For $k = 1 To $g_aTrDepth[$iNode]
        $sPad &= "    "
    Next
    Local $s = $sPad & Tree_Name($iNode) & "   kids " & Tree_NChildren($iNode)
    If $sData <> "" Then $s &= "   " & $sData
    ConsoleWrite($s & @CRLF)
    Return 0
```

THE TREE OF CHAPTER 01, PREORDER

```
the whole tree, preorder:
Photos   kids 1
    2026   kids 2   album 2026
        may   kids 2   May trip
            img0001   kids 0   1.4 MB
            img0002   kids 0   2.0 MB
        june   kids 1   June party
            img0003   kids 0   3.1 MB
```

A callback has to be a top-level function: AutoIt cannot call a function declared inside another one. Returning a negative number stops the walk and Tree_Walk reports how many nodes it visited in @extended.

The fast path, if a loop runs a hundred thousand times

An API call spends most of its time checking its arguments and setting @error. When you are inside a hot loop, read the arrays themselves - they are public globals, prefixed $g_aTr and $g_iTr, and a read costs about a fifth of the API price. The same query, 200 000 times on 20 000 nodes, in one run:

WAY TO ASK	COST PER READ	WHAT IT IS
Tree_Parent($i)	15.7 us	checks the id, sets @error, returns
$g_aTrParent[$i]	5.4 us	one integer array read, no check

DO NOT MIX THE TWO BY ACCIDENT

The globals are the UDF's own memory. Reading them is documented and safe. Writing them is not: link a parent by hand and the child count, the sibling chain and the preorder index all disagree with the tree, and only Tree_Validate will tell you. Change shape through the API, read through the globals.

04

SHAPE

Growing and reshaping

Four calls change the tree: Tree_Add, Tree_AddPath, Tree_Move and Tree_Remove. The fourth thing this chapter adds is one rule about the index.

Adding

Tree_Add($iParent, $vData, $sName) hangs one node and returns its id. The name is optional, and if you give one it must not already be used - a duplicate answers 0 with error 6. Tree_AddPath($sPath, $sSep, $vData) walks a separator-delimited chain from the root, creating only the segments that are missing, and tells you in @extended whether everything already existed.

```
Tree_Create("Photos")
Local $y = Tree_Add(Tree_Root(), "album 2026", "2026")
Local $may = Tree_Add($y, "May trip", "may")
Local $img1 = Tree_Add($may, "1.4 MB", "img0001")
Tree_Add($may, "2.0 MB", "img0002")
Local $june = Tree_Add($y, "June party", "june")
Tree_Add($june, "3.1 MB", "img0003")
Local $s = "live " & Tree_Count() & "  capacity " & Tree_Capacity()
$s &= "  depth of june " & Tree_Depth($june)
ConsoleWrite($s & @CRLF)
```

BUILD BLOCK OUTPUT

```
live 7  capacity 1024  depth of june 2
```

```
Local $wed = Tree_AddPath("2026\July\wedding", "\", "album")
ConsoleWrite("AddPath -> id " & $wed & ", all existed? " & @extended & @CRLF)
ConsoleWrite("depth " & Tree_Depth($wed) & ", 2026 now has " & Tree_NChildren($y) & " children" & @CRLF)
```

ADDPATH BLOCK OUTPUT

```
AddPath -> id 9, all existed? 0
depth 3, 2026 now has 3 children
```

capacity 1024 is the initial capacity asked for by Tree_Create; the arrays double when they run out, so you never reserve anything, but see chapter 07 if you know you are loading a hundred thousand nodes.

Removing, three ways

TREE_REMOVE($I, MODE)	WHAT HAPPENS TO ITS CHILDREN	RETURNS	FAILS
0 (default)	nothing moves: if there is at least one child the call refuses	1	error 5, child count in @extended
1 promote	children take the grandparent's place, in order	1	error 1 or 8
2 subtree	the node and everything under it are deleted	nodes deleted	error 1 or 8

Removing the root is refused with error 9 in every mode: use Tree_Create if you want an empty tree. A deleted id comes back: the free list is last-in-first-out, so the next Tree_Add usually reuses the id you just freed.

```
Local $killed = Tree_Remove($may, 2)
ConsoleWrite("Remove(may, 2) deleted " & $killed & " nodes, live now " & Tree_Count() & @CRLF)
$s = "Find(may) -> " & Tree_Find("may") & " (error " & @error & ")"
$s &= ", stale keys " & Tree_NameStale()
ConsoleWrite($s & @CRLF)
ConsoleWrite("CompactNames -> " & Tree_CompactNames() & " names left" & @CRLF)
```

REMOVE BLOCK OUTPUT

```
Remove(may, 2) deleted 2 nodes, live now 7
Find(may) -> 0 (error 1), stale keys 1
CompactNames -> 7 names left
```

Moving, and the one rule

```
Tree_Move($img1, $wed)
$s = "before rebuild: clean " & Tree_IsClean()
$s &= ", wedding subtree " & Tree_SubtreeSize($wed) & " (walked)"
ConsoleWrite($s & @CRLF)
Tree_Rebuild()
$s = "after rebuild: clean " & Tree_IsClean()
$s &= ", wedding subtree " & Tree_SubtreeSize($wed) & ", live " & Tree_Count()
ConsoleWrite($s & @CRLF)
```

MOVE BLOCK OUTPUT

```
before rebuild: clean 0, wedding subtree 2 (walked)
after rebuild: clean 1, wedding subtree 2, live 9
```

1. **Any add, move or remove marks the preorder index dirty.** Tree_IsClean() goes to 0 and stays 0.
2. **While it is dirty, subtree queries are correct but slow.** Tree_SubtreeSize walks that subtree instead of reading one number.
3. **When you are done changing the shape, call Tree_Rebuild() once.** One pass over every node, about 14 microseconds per node.
4. **Do not rebuild after every single change.** That is the mistake that makes a tree feel slow: rebuild at the end of the batch.

> **THE INTERVAL IS THE THING TO REMEMBER**
>
> While the index is clean, a subtree is a contiguous range in the preorder array, which is what makes Tree_SubtreeSize answer in one read and Tree_Remove free a whole subtree in one pass. After a change, that range may be wrong. Nothing in the UDF will ever give you a wrong answer - Tree_SubtreeSize detects the dirty index and walks - but any range you computed yourself before a change is stale.

How Tree_SubtreeSize answered

@EXTENDED	IT DID	COST
1	read one number from a clean index	O(1)
2	walked that subtree because the index is dirty	proportional to the subtree

05

THE INDEX

Names, and why deleting stays fast

A name is an index, not the data. This chapter is short, but it is the reason this UDF keeps its speed when you delete things.

The problem it solves

Without an index, finding a node by name means looking at every node until one matches. On 20 000 nodes, 200 name searches cost 10 961 ms in the forum idiom, because that idiom scans: 54.8 ms for one answer. The same 2 000 searches through Tree_Find cost 69 ms, or 34.7 us each.

The one thing that was slow, and what changed

A Map is not constant time. Removing a key costs more as the Map grows, and the probe says so plainly:

NAMED NODES	INSERT	READ	MAPREMOVE	MAPKEYS
10 000	2 us	2 us	18 us	under 1 us per key
25 000	2 us	3 us	50 us	under 1 us per key
50 000	5 us	8 us	117 us	under 1 us per key
100 000	21 us	28 us	438 us	under 1 us per key

So Tree_Add puts the name in the Map, and Tree_Remove takes nothing out. It only counts the key as stale. Two things then clean up after it, and neither is on the delete path:

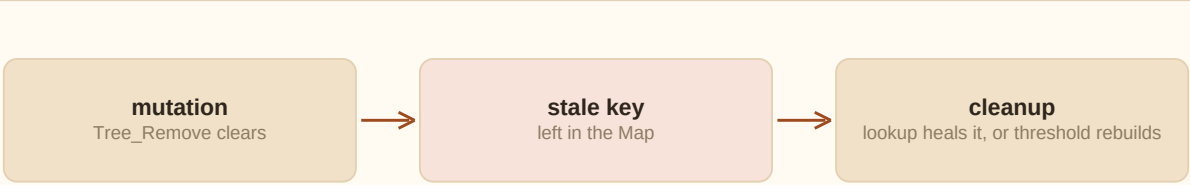


Figure 2. *The delete path touches nothing in the Map. The two arrows to the right are the only places the Map is cleaned, and neither runs while you are deleting.*

- **Heal on lookup.** Tree_Find checks that the id the Map gave really is alive and really carries that name. If not, the key is dropped and the answer is "not found", so a stale key can never return a wrong node.
- **Rebuild by threshold.** Past 4 096 stale keys - and past 65 536 whenever they are under a quarter of the live nodes - the Map is thrown away and rebuilt in one pass. Emptying a Map by reassignment costs 5.5 ms for 20 000 keys; the same thing done with MapRemove in a loop cost 509 ms.

What it bought, measured

ONE SUBTREE DELETE	PER NODE DELETED	SOURCE
0.1.0, MapRemove on every delete	121.3 us	Tree_Bench_0.1.0.log
0.2.0, lazy invalidation	65.3 us	Tree_Bench.log
0.2.0 with no names at all	37.5 us	the floor to aim at

The price paid is that a name lookup now verifies what it found: 34.7 us per Tree_Find against 24.7 us in 0.1.0. On the same run, deleting a named subtree dropped by more than half.

Reading the transcript line from chapter 04

It said: Find(may) -> 0 (error 1), stale keys 1. Two nodes died in that delete, so two keys were stale, and the number printed 1 because Tree_Find("may") on the same line healed that key before Tree_NameStale() was read. The order inside one ConsoleWrite expression is left to right: the lookup came first, the count second.

- **Tree_NameStale()** an upper bound on stale keys, not a promise.
- **Tree_CompactNames()** forces the rebuild now and returns the live name count.
- **Tree_UseNameIndex(False)** turns the index off: names are still stored, only the Map is dropped, and Tree_Find answers error 9. Past about 50 000 named nodes this is the faster setting.

> **IF YOU NEVER DELETE**
> A tree you only build and read never grows stale keys, and this whole chapter is a design note rather than something you must act on. It becomes real when a script deletes tens of thousands of named nodes in one run.

06

TROUBLE

When it goes wrong

Every call sets @error to one of ten numbers and returns something sensible for it: usually 0, sometimes a count. Nothing throws, nothing dialogs, nothing exits.

The ten numbers

CONSTANT	VALUE	MEANS	WHAT TO CHECK FIRST
$TREE_E_NONE	0	success	-
$TREE_E_NODE	1	unknown, dead or out-of-range id	was that node deleted, or is the id past Tree_Capacity()?
$TREE_E_PARENT	2	a parent link points at a dead node	Tree_Validate: this means the structure is broken, report it
$TREE_E_MEM	3	the 4 000 000 id ceiling is reached	fewer nodes, or a second run
$TREE_E_CYCLE	4	a move that would put a node under itself	check the target is not inside the node you are moving
$TREE_E_NOTEMPTY	5	the node has children and mode 0 refused	@extended carries the child count; ask the user, or use mode 2
$TREE_E_NAME	6	that name is already used	Tree_Find it first, or give the node no name
$TREE_E_FILE	7	export or import could not be done	for import, @extended is the offending line number
$TREE_E_STATE	8	no tree created yet	call Tree_Create once at the start
$TREE_E_ARG	9	bad argument, or a broken link chain	a name index turned off, an empty path, a root delete

Eight things that look like bugs

- **Tree_Find returns 0 for a name you are sure you set.** The node was deleted, or renamed, or you typed the name with different case: names are compared exactly.
- **Tree_Add answers 0 with error 6.** The name is taken somewhere in the tree, not just under that parent. Names are unique globally.
- **Tree_Add answers 0 with error 9.** The name index is switched off and you passed a name. Pass "" or turn the index back on.
- **Tree_SubtreeSize is right but slow.** The index is dirty and the call walked. Tree_Rebuild once, then ask again.
- **An id you stored came back with error 1.** It was deleted and recycled. Do not keep ids across deletes; keep names and re-Find.
- **Tree_Import refuses a file Tree_Export wrote.** @extended gives the line number. The root must be the first row, and every parent must already be above it.
- **A callback that calls Tree_Remove stops the walk oddly.** It is not a bug: the walk cursor follows links that the delete just rewrote. Collect ids in the callback, change the shape after the walk.

- **Nothing prints, exit code 1.** You ran AutoIt3.exe with a switch it does not have, such as /run, or the script never reached a ConsoleWrite.

> **NOT REENTRANT**
> Five routines share one scratch stack: Tree_Rebuild, Tree_Remove, Tree_SubtreeSize, Tree_Validate and Tree_Move. Calling one of them from inside a function that is already using it corrupts the walk. Keep your Tree_Walk callback read-only, and never drive the UDF from a GUI or timer callback that can be re-entered while another call is still running.

Hard limits, stated plainly

- **One tree per script** because the state is global. Tree_Create wipes what is live.
- **4 000 000 nodes** the id ceiling. Above it, Tree_Add answers error 3.
- **ids are recycled** a deleted id is usually the next one handed out. Never trust a stored id after a delete.
- **depth is not free** Tree_Move refreshes the stored depth of the whole moved subtree, so moving a big subtree costs what it is.
- **arrays double, they never shrink** Tree_Capacity() only goes up. If you load a million and drop to ten, memory stays reserved until Tree_Create.

NEXT CHAPTER 07 The whole surface, and the proof

07

THE WHOLE SURFACE

The whole surface, and the proof

Thirty-one public calls, then the numbers behind the claims in this booklet, then how to make this PDF again.

Every public call

CALL	RETURNS	SETS @ERROR TO
Tree_Create($sName, $iCap)	capacity allocated	0
Tree_Root() / Tree_Count() / Tree_Capacity()	root id / live nodes / slots	0 / 0 / 0
Tree_IsClean() / Tree_NameStale()	index valid / stale keys	0 / 0
Tree_CompactNames()	live names after rebuild	0
Tree_UseNameIndex($b)	1 when accepted	0 or 3
Tree_Add($iParent, $vData, $sName)	new id	0, 1, 3, 6, 8, 9
Tree_AddPath($sPath, $sSep, $vData)	last id, @extended exact	0, 1, 3, 6, 8, 9
Tree_SetData($i, $v) / Tree_GetData($i)	1 / the value	0 or 1
Tree_SetName($i, $sName)	1	0, 1, 6, 9
Tree_Name($i) / Tree_Parent($i) / Tree_Depth($i)	name / parent / depth	0 or 1
Tree_NChildren($i)	child count	0 or 1
Tree_Children($i, $aOut)	count, fills 1-based array	0 or 1
Tree_FirstChild($i) / Tree_NextSibling($i)	id or 0	0 or 1
Tree_IsAncestor($a, $i, $bSelf)	1 or 0	0 or 1
Tree_Find($sName)	id or 0	0 or 1
Tree_Remove($i, $iMode)	nodes deleted	0, 1, 5, 8, 9
Tree_Move($i, $iNewParent)	1	0, 1, 4, 8, 9
Tree_SubtreeSize($i)	count, @extended 1 or 2	0 or 1
Tree_Rebuild()	nodes reached	0, 3, 8
Tree_Walk($sFunc, $iNode)	nodes visited	0, 1, 8, 9
Tree_PathTo($i, $aOut)	steps, fills 1-based array	0 or 1
Tree_Info($sKey)	one number, key in: "count ids capacity free dirty clean root maxdepth arrays cells"	0 or 9
Tree_Validate()	1 if the whole structure agrees	0, 2, 4, 9
Tree_Export($sFile) / Tree_Import($sFile)	nodes written / nodes read	0, 7, 8

One tree, 20 000 nodes, one run, four shapes

QUESTION	THIS UDF	AUTOITOBJECT IDIOM	MAP OF CHILD ARRAYS	NOTE
build the tree	1 437 ms	6 932 ms	461 ms	ms, 20 000 nodes
read one parent	15.7 us	10.5 us	7.7 us	API call, 200 000 reads
same read, raw arrays	5.4 us	no arrays to read	not offered	the fast path
subtree size	14.8 us	125.0 us	not offered	2 000 queries
find by name	34.7 us	54 804.1 us	not offered	scan against index
reorder the shape	129.7 us	82.8 us	206.1 us	2 000 moves
memory per node	493 B	not measured	606 B	50 000 nodes, structure only
one preorder pass	277 ms	280 ms	not offered	20 000 nodes

Read the two columns that go against this UDF as well: the object idiom wins a single property read and a cheap re-link, and a Map of child arrays builds faster because it maintains nothing. What it cannot do is find a name, size a subtree, or keep that speed once nodes are deleted.

Two more measurements, same discipline

WHAT	MEASURED	WHY IT MATTERS
100 000 nodes built from a 1 024 capacity	6 119 ms against 4 677 ms	pass Tree_Create("x", n) when you know n: 24% off
deleting 20 000 nodes, named against anonymous	1 871 ms against 1 777 ms	the index costs a few microseconds per node, not a hundred
emptying a 20 000-key Map, two ways	5.5 ms against 509 ms	why the rebuild is an assignment, not a loop
assertions run against this build	5 162 passed, 0 failed	Tree_SelfTest.au3, seed 20261112

> **ABOUT THESE NUMBERS**
> This machine has 3.8 GB of memory with only a few hundred MB free, so the same row can differ by a factor of two between two runs of the same script. Never compare one log against another for a small difference; only ratios inside a single run are worth repeating. Rerun and compare the logs you get with the ones next to the source.

The files, and how to rebuild this booklet

FILE	WHAT IT IS	RUN IT WITH
Tree.au3	the UDF. The only file your script needs.	#include "Tree.au3"
Tree_Starter.au3	the twelve-line program of chapter 02.	AutoIt3.exe Tools\Tree_Starter.au3

FILE	WHAT IT IS	RUN IT WITH
Tree_QuickStart_Sample.au3	every block quoted in this booklet.	AutoIt3.exe Tools\Tree_QuickStart_Sample.au3
Tree_Example.au3	the whole surface in eight commented sections.	AutoIt3.exe Tree_Example.au3
Tree_SelfTest.au3	the assertion suite, plus a randomised one.	AutoIt3.exe Tree_SelfTest.au3 400 6000
Tree_Bench.au3	the four shapes of chapter 07, and modes mem, map, grow.	AutoIt3.exe Tree_Bench.au3
Tree_CostProbe.au3	the primitive costs of chapter 05.	AutoIt3.exe Tree_CostProbe.au3
Tools\build_quickstart.au3	this PDF, generated from the files above.	AutoIt3.exe Tools\build_quickstart.au3

Licence MIT. The booklet itself is typeset by PDFX 1.4.0, and its source is Tools\build_quickstart.au3 plus Tools\quickstart_content.au3: nothing in it is typed by hand except the prose, so the code blocks and the tables cannot drift from what was tested.