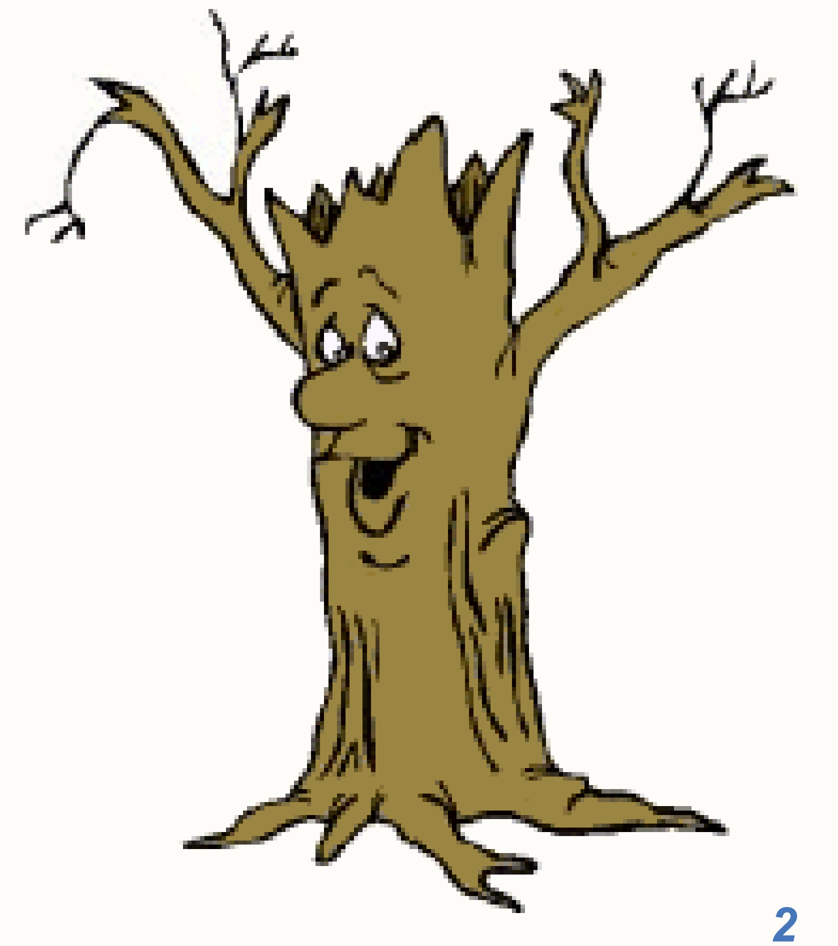
# 第三章：实验统计分析工具ROOT

授课人：董燎原
中国科学院高能物理研究所

# I/O and Trees



# Outline

- Introduction to I/O in ROOT
  - how to save ROOT objects in a file
  - example: saving an histogram
- ROOT Trees:
  - `TNtuple` class ( a simple Tree)
  - `TTree` class
- How to create a Tree and to write in a file
- Merging of Trees: `TChain`
- Using Tree Friends
- How to read and analyze the Tree

# Input and Output

参考 https://root.cern.ch/input-and-output

# Saving Objects in ROOT

- Use the `TFile` class
  - We need first to create the class, which opens the file

```
TFile* f = TFile::Open("file.root","NEW");
```

use option "`RECREATE`" if the file already exists

- Write an object deriving from `TObject`:

```
object->Write("optionalName")
```

if the optionalName is not given the object will be written in the file with its original name (`object->GetName()`)

- For objects that do not inherit from `TObject`, use :

```
f->WriteObject(object, "name");
```

# TFile Class

- ROOT stores objects in TFiles:

```
TFile* f = TFile::Open("file.root", "NEW");
```

- TFile behaves like file system:

```
f->mkdir("dir");
```

- TFile has a current directory:

```
f->cd("dir");
```

- You can browse the content:

```
f->ls();
 TFile**          file.root
  TFile*          file.root
   TDirectoryFile*          dir   dir
   KEY: TDirectoryFile dir;1 dir
```

# Saving Histogram in a File

- How to save objects in a file

```
TFile* f = TFile::Open("myfile.root","NEW");
TH1D* h1 = new TH1D("h1", "h1",100,-5.,5.);
h1->FillRandom("gaus"); // fill histogram with random data
h1->Write();
delete f;
```

- TFile compresses data using ZIP

```
h1->Write();
f->GetCompressionFactor()
(Float_t)2.34518527984619141e+00
```

# Where is My Histogram ?

- All histograms and trees are owned by `TFile` which acts like a scope
- After closing the file (i.e when the file object is deleted) also the histogram, trees and graphs objects are deleted
- This code will crash ROOT:

```
TFile* f = TFile::Open("myfile.root","RECREATE");

TH1D* h1 = new TH1D("h1","h1",100,-5.,5.);
delete f;
h1->Draw(); // will crash - DO NOT DO IT!!!
*** Break *** segmentation violation
```

- Other objects will be still there and can be accessed afterwards
- This can be changed with `TH1::AddDirectory(false);`

# Reading a File

- Reading is simple:

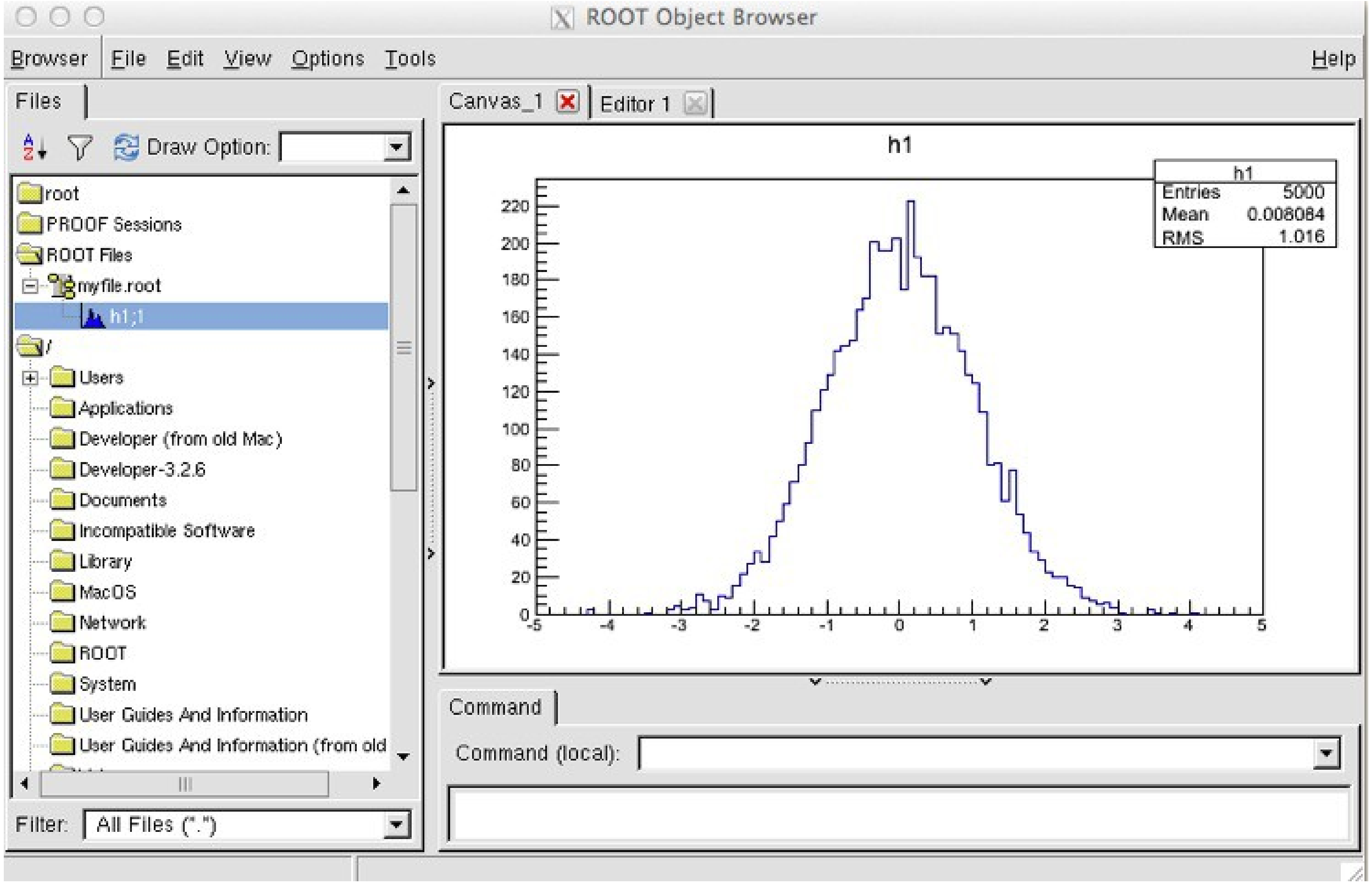
```
TFile*  f = TFile::Open("myfile.root");
TH1* h1 = 0;
f->GetObject("h1",h1);
h1->Draw();
Delete f;
```

- Can also use
  - `TH1 * h = (TH1*) f->Get("h1");`
  - `TH1 * h = (TH1*) f->GetObjectChecked("h1","TH1");`
    - which returns a null pointer if the read object is not of the right type
- Remember:
  - TFile owns the histogram
  - the histogram is gone when the file is closed
  - to change this add `TH1::AddDirectory(false)` in `root_logon.C`

# TBrowser

- GUI for browsing ROOT objects written in a file

```
root [0] new TBrowser();
```



# Merging ROOT Files

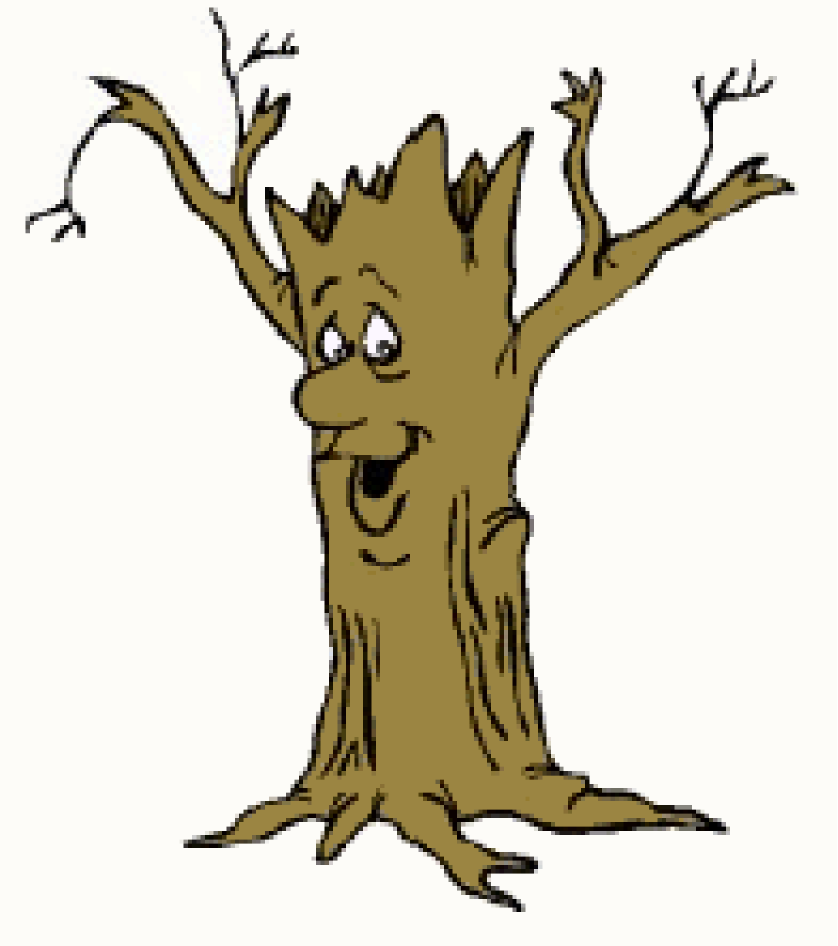
To merge Root files containing histograms or/and Trees,
use the utility hadd in $ROOTSYS/bin/hadd.

At the shell command line, simply type hadd to get online help.

例如：
hadd  out.root  file1.root file2.root ... filen.root
将files1.root, file2.root, ....., filen.root 加到 out.root 中。

或者用通配符:
Hadd  out.root  file*.root

# Trees



参考 https://root.cern.ch/root/htmldoc/guides/users-guide/ROOTUsersGuide.html#trees
tutorials: https://root.cern.ch/doc/master/group__tutorial__tree.html

# Why Should You Use a Tree ?

- In case you want to store large quantities of same-class objects, ROOT has designed the TTree and TNtuple classes:
  - The **TTree** class is optimized to reduce disk space and enhance access speed
  - A **TNtuple** is a **TTree** that is limited to only hold floating-point numbers
  - a **TTree** can hold all kind of data, such as objects or arrays in addition to all the simple types.

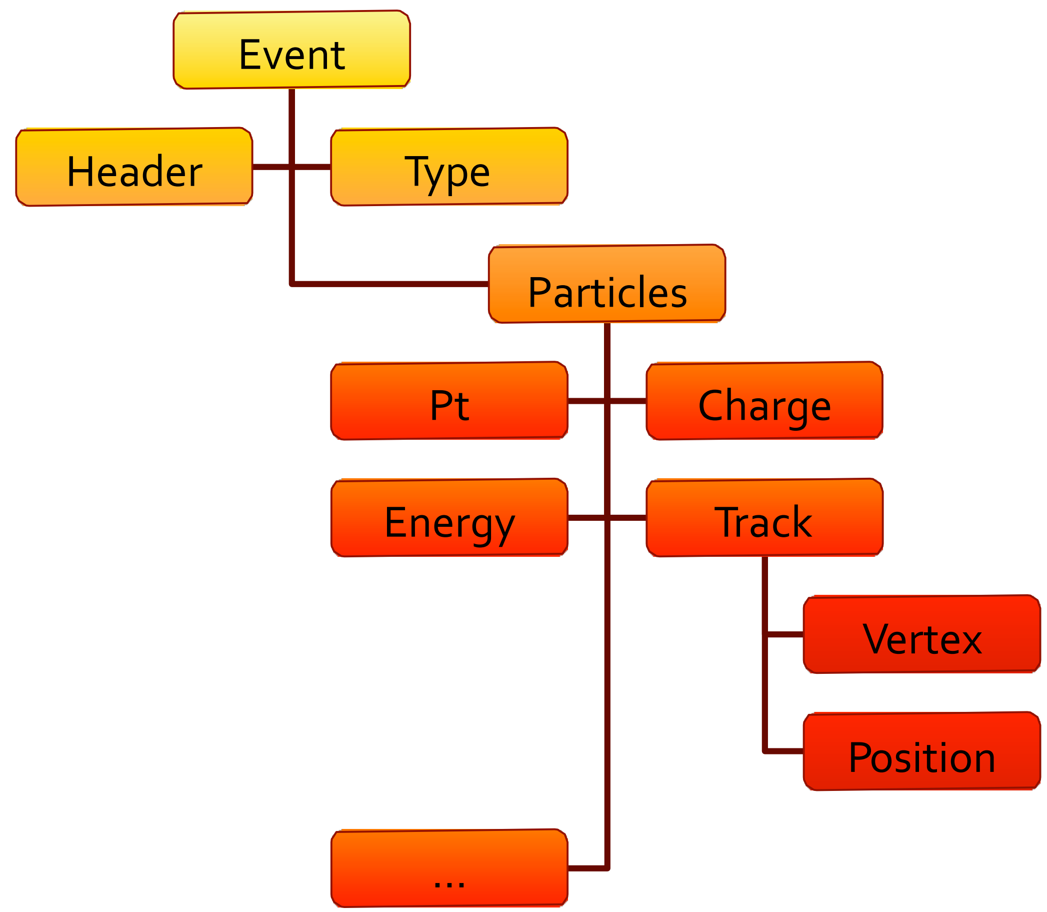
- When using a **TTree**, we fill its branch buffers and the buffers are written to disk when it is full.
  - **TTree** takes advantage of compression when the objects are written a bunch at a time.
  - TTree reduces the header of each object
  - TTree optimizes the data access

# Ntuple and Trees

- Ntuple class:
  - TNtuple
    - for storing tabular data
    - e.g. Excel Table with numbers

| x | y | z |
|---|---|---|
| -1.10228 | -1.79939 | 4.452822 |
| 1.867178 | -0.59662 | 3.842313 |
| -0.52418 | 1.868521 | 3.766139 |
| -0.38061 | 0.969128 | 1.084074 |
| 0.552454 | -0.21231 | 0.350281 |
| -0.18495 | 1.187305 | 1.443902 |
| 0.205643 | -0.77015 | 0.635417 |
| 1.079222 | -0.32739 | 1.271904 |
| -0.27492 | -1.72143 | 3.038899 |
| 2.047779 | -0.06268 | 4.197329 |
| -0.45868 | -1.44322 | 2.293266 |
| 0.304731 | -0.88464 | 0.875442 |
| -0.71234 | -0.22239 | 0.556881 |
| -0.27187 | 1.181767 | 1.470484 |
| 0.886202 | -0.65411 | 1.213209 |
| -2.03555 | 0.527648 | 4.421883 |
| -1.45905 | -0.464 | 2.344113 |
| 1.230661 | -0.00565 | 1.514559 |
| [illegible] | [illegible] | 3.562347 |

- Tree class
  - TTree
    - for storing complex data types
    - e.g. DataBase tables

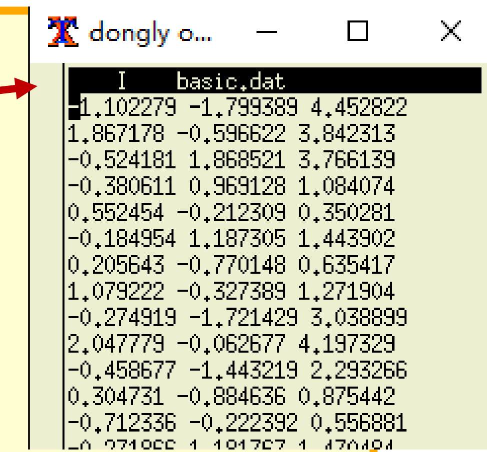


# Building ROOT Ntuple

*root $ROOTSYS/tutorials/tree/basic.C*
*也可参考 basic2.C*

- Creating and Storing N-tuples
  - The ROOT class `TNtuple` can store only floating entries
    - each raw (record) must be composed only of floating types
  - Specify the name (label) of the type when creating the object

```
#include "Riostream.h"
void basic() {
ifstream in;
   in.open(Form("basic.dat"));
   Float_t x,y,z; Int_t nlines = 0;
   TFile *f = new TFile("basic.root","RECREATE");
   TH1F *h1 = new TH1F("h1","x distribution",100,-4,4);
   TNtuple *ntuple = new TNtuple("ntuple","data from ascii file","x:y:z");
   while (1) {
      in >> x >> y >> z;
      if (!in.good()) break;
      if (nlines < 5) printf("x=%8f, y=%8f, z=%8f\n",x,y,z);
      h1->Fill(x);
      ntuple->Fill(x,y,z);
      nlines++;
   }
   printf(" found %d points\n",nlines);
   in.close();
   f->Write();
}
```



# How To Read a NTuple

- Open the file and get the ntuple object

```
TFile f("basic.root");
ntuple->Print();
```

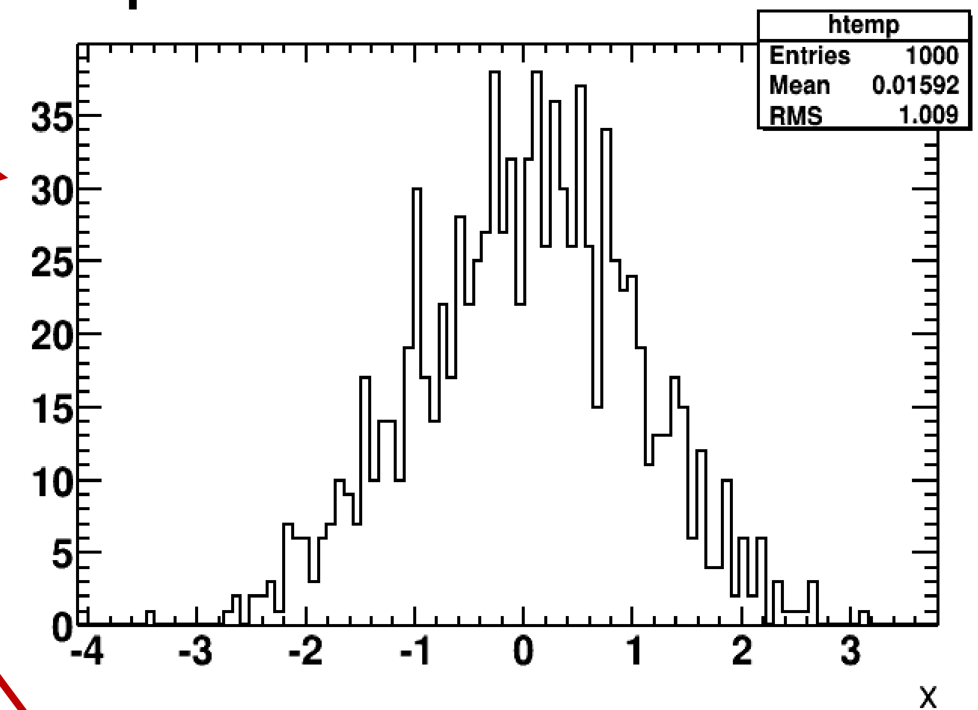
Note that (as for histograms) we do not need to use TFile::Get
This works only in CINT, not valid C++

```
Welcome to ROOT!
root [0] TFile f("basic.root");
root [1] ntuple->Print();
******************************************************************************
*Tree    :ntuple    : data from ascii file                                   *
*Entries :     1000 : Total =           13952 bytes  File  Size =      11902 *
*        :          : Tree compression factor =   1.07                       *
******************************************************************************
*Br    0 :x         : Float_t                                                *
*Entries :     1000 : Total  Size=       4526 bytes  File Size  =       3824 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=   1.06     *
*............................................................................*
*Br    1 :y         : Float_t                                                *
*Entries :     1000 : Total  Size=       4526 bytes  File Size  =       3826 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=   1.06     *
*............................................................................*
*Br    2 :z         : Float_t                                                *
*Entries :     1000 : Total  Size=       4526 bytes  File Size  =       3754 *
*Baskets :        1 : Basket Size=      32000 bytes  Compression=   1.08     *
*............................................................................*
root [2]
```

# Looking at the Ntuple

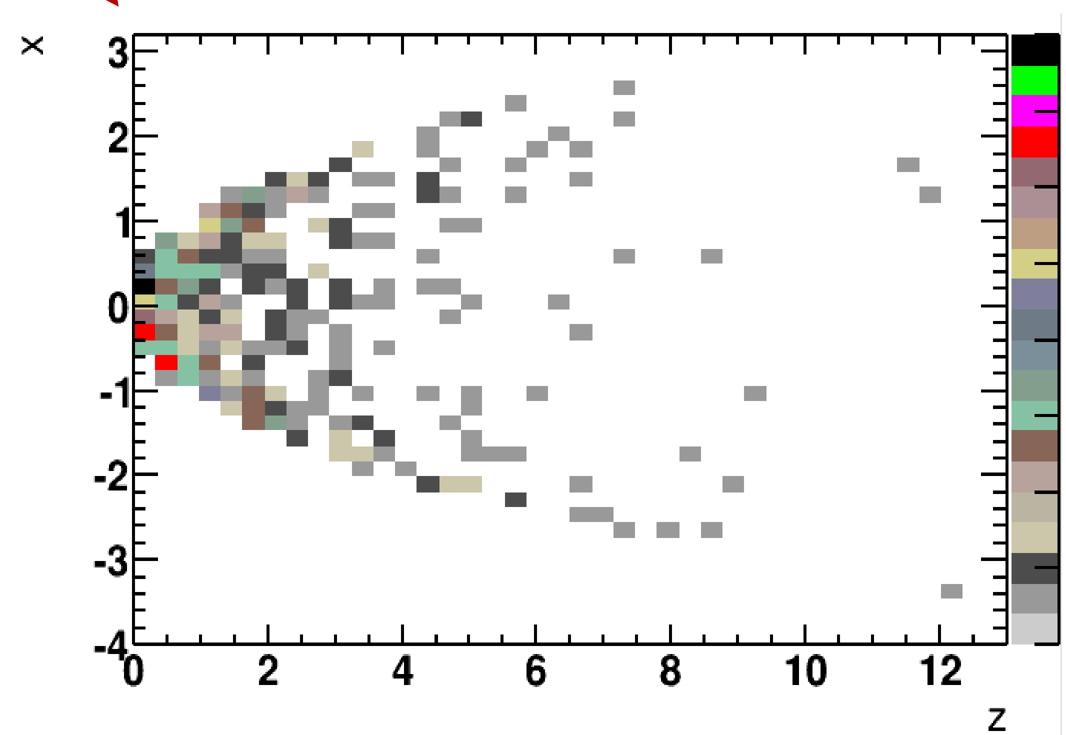
- Can Draw one of the variable of the ntuple:

```
ntuple->Draw("x")
```




- Can Draw 2 (or more) variables:

```
ntuple->Draw("x:z","y>0","colz")
```




- Can Scan the variables' values:

```
ntuple->Scan("x:y:z")
```

```
root [4] ntuple->Scan("x:y:z")
************************************************
*    Row   *         x *         y *         z *
************************************************
*        0 * -1.102278 * -1.799389 * 4.4528222 *
*        1 * 1.8671779 * -0.596621 * 3.8423130 *
*        2 * -0.524181 * 1.8685209 * 3.7661390 *
*        3 * -0.380611 * 0.9691280 * 1.0840740 *
*        4 * 0.5524539 * -0.212309 *  0.350281 *
*        5 * -0.184954 * 1.1873049 * 1.4439020 *
*        6 * 0.2056429 * -0.770147 * 0.6354169 *
*        7 * 1.0792219 * -0.327389 * 1.2719039 *
*        8 * -0.274919 * -1.721428 * 3.0388989 *
*        9 * 2.0477790 * -0.062677 * 4.1973290 *
*       10 * -0.458676 * -1.443218 * 2.2932660 *
*       11 * 0.3047310 * -0.884635 * 0.8754420 *
*       12 * -0.712336 * -0.222391 * 0.5568810 *
*       13 * -0.271865 * 1.1817669 * 1.4704840 *
*       14 * 0.8862019 * -0.654106 * 1.2132090 *
*       15 * -2.035552 * 0.5276479 * 4.4218831 *
*       16 * -1.459046 * -0.463997 * 2.3441131 *
*       17 * 1.2306610 * -0.005650 * 1.5145590 *
*       18 * 0.0887869 * 1.8853290 * 3.5623469 *
*       19 * -0.314153 * -0.329160 * 0.2070399 *
*       20 * -0.198253 * 0.6460700 * 0.4567120 *
*       21 * -1.636217 * 1.0495510 * 3.7787621 *
*       22 * 1.2211090 * 0.8143829 * 2.1543269 *
*       23 * 1.4131350 * 1.5498369 * 4.3989419 *
*       24 * -0.174493 * -1.330937 * 1.8018410 *
Type <CR> to continue or q to quit ==>
```

# Getting The Entries

- Entries of a ROOT N-tuple can be retrieved using `TNtuple::GetEntry(irow)`

```
Tfile f(“basic.root");

TNtuple *ntuple=0;
f.GetObject("ntuple",ntuple);

// loop on the ntuple entries
for (int i = 0; i < ntuple->GetEntries(); ++i) {

  ntuple->GetEntry(i);
  float * raw_content = ntuple->GetArgs();
  float     x     =     raw_content[0];
  float     y     =     raw_content[1];
  float     z     =     raw_content[2];

  // do something with the data..
}
```

# ROOT Data Format - Tress

- ROOT N-tuple can store only floating point variables
- For storing complex types, i.e. objects we can use the ROOT tree class, TTree
  - TNtuple is a special case of a TTree (a derived class)

- The ROOT Tree is
  - Extremely efficient write once, read many.
  - Designed to store >$10^9$ (HEP events).
  - Trees allow fast direct and random access to any entry (sequential access is the best).
  - Trees are build with “branches” and “leaves”. One can read a subset of all branches.
  - Optimized for network access (read-ahead).

# Why Trees ?

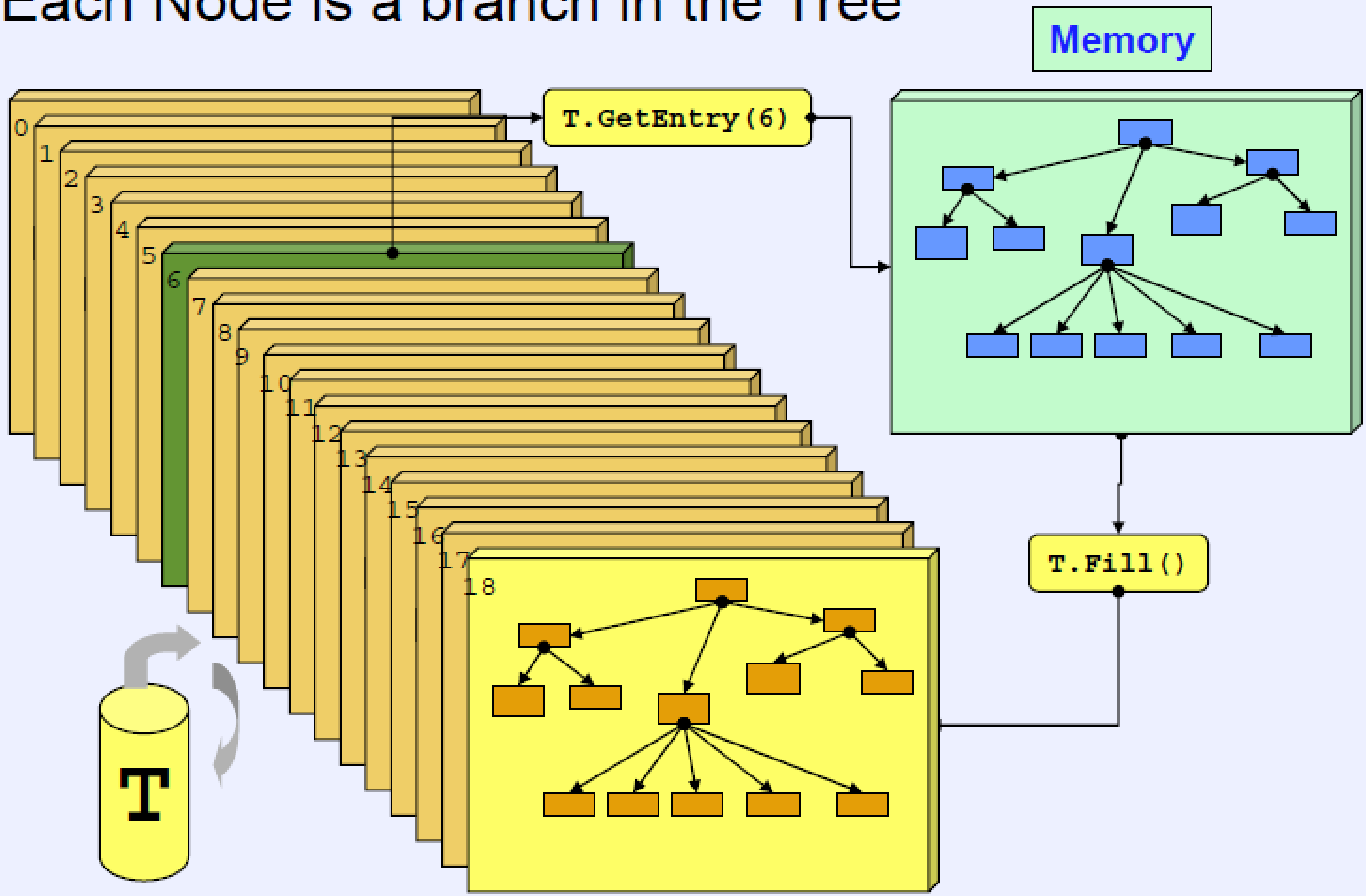
- object.Write() is convenient for simple objects like histograms, but inappropriate for saving collections of events containing complex objects
- High level functions like TTree::Draw loop on all events with selection expressions.
- Reading a collection:
  - read all elements (all events)
- With trees:
  - only one element in memory,
  - or even only a part of it (less I/O)
- Trees buffered to disk (TFile);
  - I/O is integral part of TTree concept

# Tree Access

- Databases have row wise access
  - Can only access the full object (e.g. full event)
- ROOT trees have column wise access
  - Direct access to any event, any branch or any leaf even in the case of variable length structures
  - Designed to access only a subset of the object attributes (e.g. only particles' energy)
  - Makes same members consecutive, e.g. for object with position in X, Y, Z, and energy E, all X are consecutive, then come Y, then Z, then E. A lot higher zip efficiency!

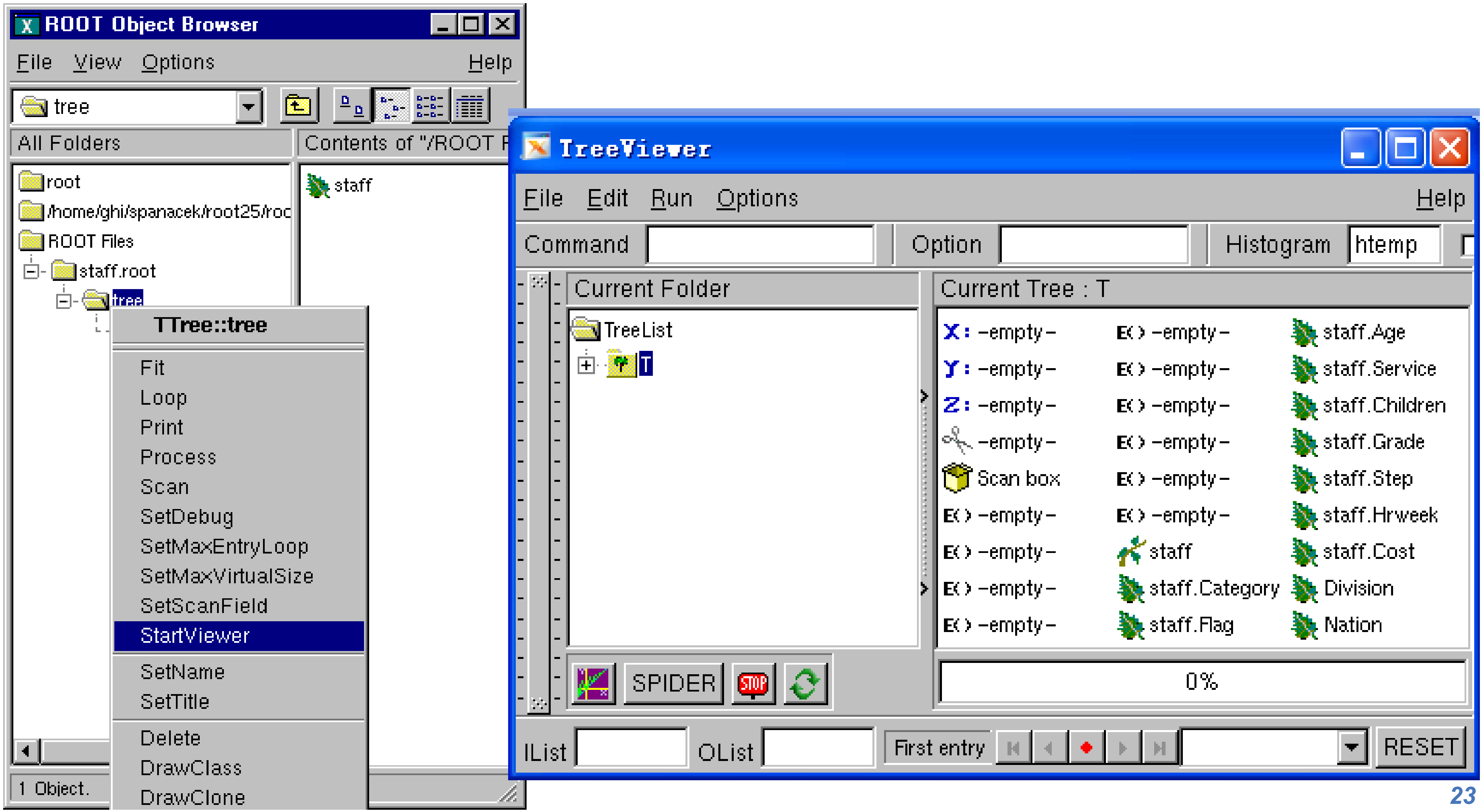
# Tree and Memory

Each Node is a branch in the Tree



# Interactive Tree Analysis

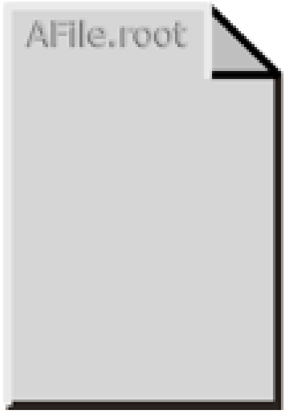
```
root[] TFile f("staff.root")
root[] T->StartViewer() //invoke the viewer by the TTree object name
//or
root[] TBrowser a //double click the root file to open
```



# Building a ROOT Tree

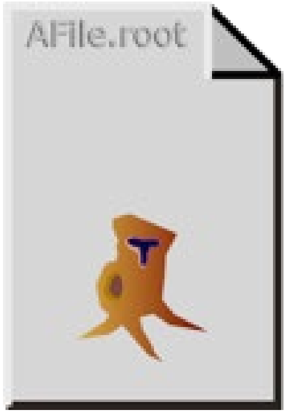
- Five steps to build a Tree
  - Create a TFile class
    - Tree can be huge → need file for swapping filled entries

```
TFile *hfile = TFile::Open("AFile.root","RECREATE");
```




  - Create a TTree class

```
TTree * tree = new TTree("myTree","A Tree");
```




  - Add a Branch (TBranch) to the TTree
  - Fill the tree with the data
  - Write the tree to file

# Tree Structure and Branches

- What is a Branch ?
  - A branch is like a directory
    - it can hold a simple variable, a list of variables, an object or evan a collection of objects
    - The leaves are the data containers of the branch
    - it is possible to read only a sub-set of all the branches in a tree
      - variables or object known to be used together should be put in the same branch
    - branches of the same tree can be written to separate files

# Adding a Branch to the Tree

**To add a branch we need**

- Name of the Branch
- Address of the pointer to the object we want to store

**To save is a list of simple variables**

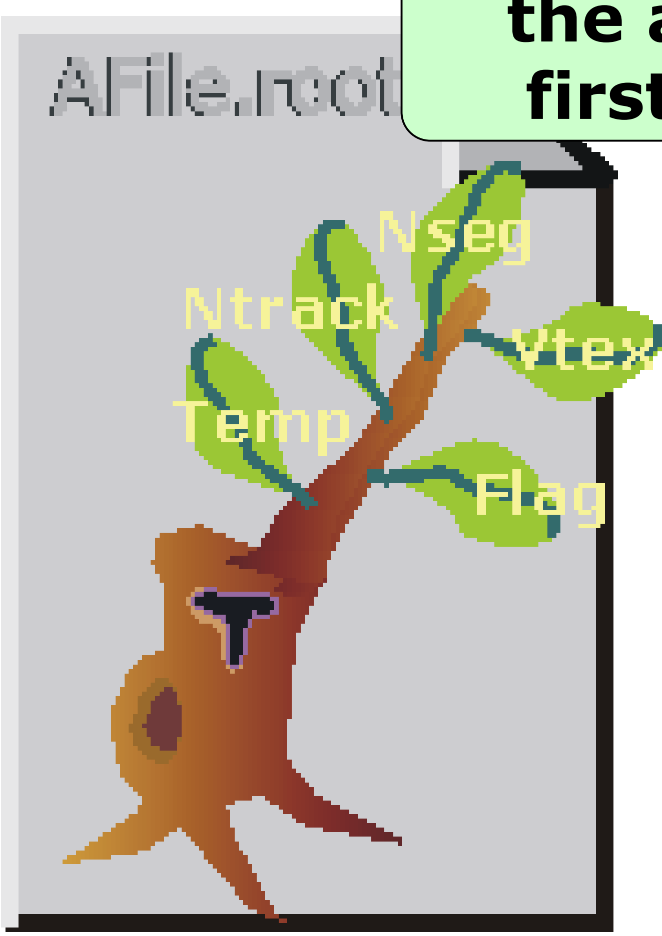
```
tree->Branch("Ev_Branch",&event,"temp/F:ntrack/I:nseg:nvtex:flag/i");
```

**Branch Name**

**a string describing the leaf list**

**the address from which the first variable is to be read**

**event** is a structure with one float and three integers and one unsigned integer




Each leaf has a name and a type (default: float) separated by a "/ " and separated from the next leaf by a ":"

```
<Variable>/<type>:<Variable>/<type>
```

The type can be omitted and if no type is given, the same type as the previous variable is assumed.

```
"ntrack/I:nseg:nvtex"
```

# Symbols Used for the Type

```
C:    a character string terminated by the 0 character
B:    an 8 bit signed integer
b:    an 8 bit unsigned integer
S:    a 16 bit signed integer
s:    a 16 bit unsigned integer
I:    a 32 bit signed integer
i:    a 32 bit unsigned integer
L:    a 64 bit signed integer
l:    a 64 bit unsigned integer
F:    a 32 bit floating point
D:    a 64 bit floating point
```

If the type consists of two characters, the number specifies the number of bytes to be used.

The line `ntrack/I2` describes *ntrack* to be written as a 16-bit integer (rather than a 32-bit integer):

# a Branch to Hold an Array

- With **TTree::Branch()** method, you can also add a leaf that holds an entire array of variables.
- To add an array of floats, use the f[n] notation when describing the leaf.

```
Float_t f[10];
tree->Branch("fBranch",f,"f[10]/F");
```

- To add an array of variable length

```
{
TFile *f = new TFile("peter.root","recreate");
Int_t nPhot;
Float_t E[500];
TTree* nEmcPhotons = new TTree("nEmcPhotons","EMC Photons");
nEmcPhotons->Branch("nPhot",&nPhot,"nPhot/I");
nEmcPhotons->Branch("E",E,"E[nPhot]/F");
}
```

*example: $ROOTSYS/tutorials/tree/tree2.C and cernstaff.C*

# a Branch to Hold an Event Object

**Example:** To write a branch to hold an event object, we need to load the object definition, e.g. the Event class in *$ROOTSYS/test/libEvent.so.*

```
root[] .L libEvent.so
```

First, we need to open a file and create a tree.

```
root[] TFile *f = new TFile("AFile.root","RECREATE")
root[] TTree *tree = new TTree("T","A Root Tree")
```

We need to create a pointer to an Event object,
Then we create a branch with the TTree::Branch method:

```
root[] Event *event = new Event()  建一个Event对象的指针
root[] tree->Branch("EventBranch","Event",&event,32000,99)
```

name of the branch, name of the class,

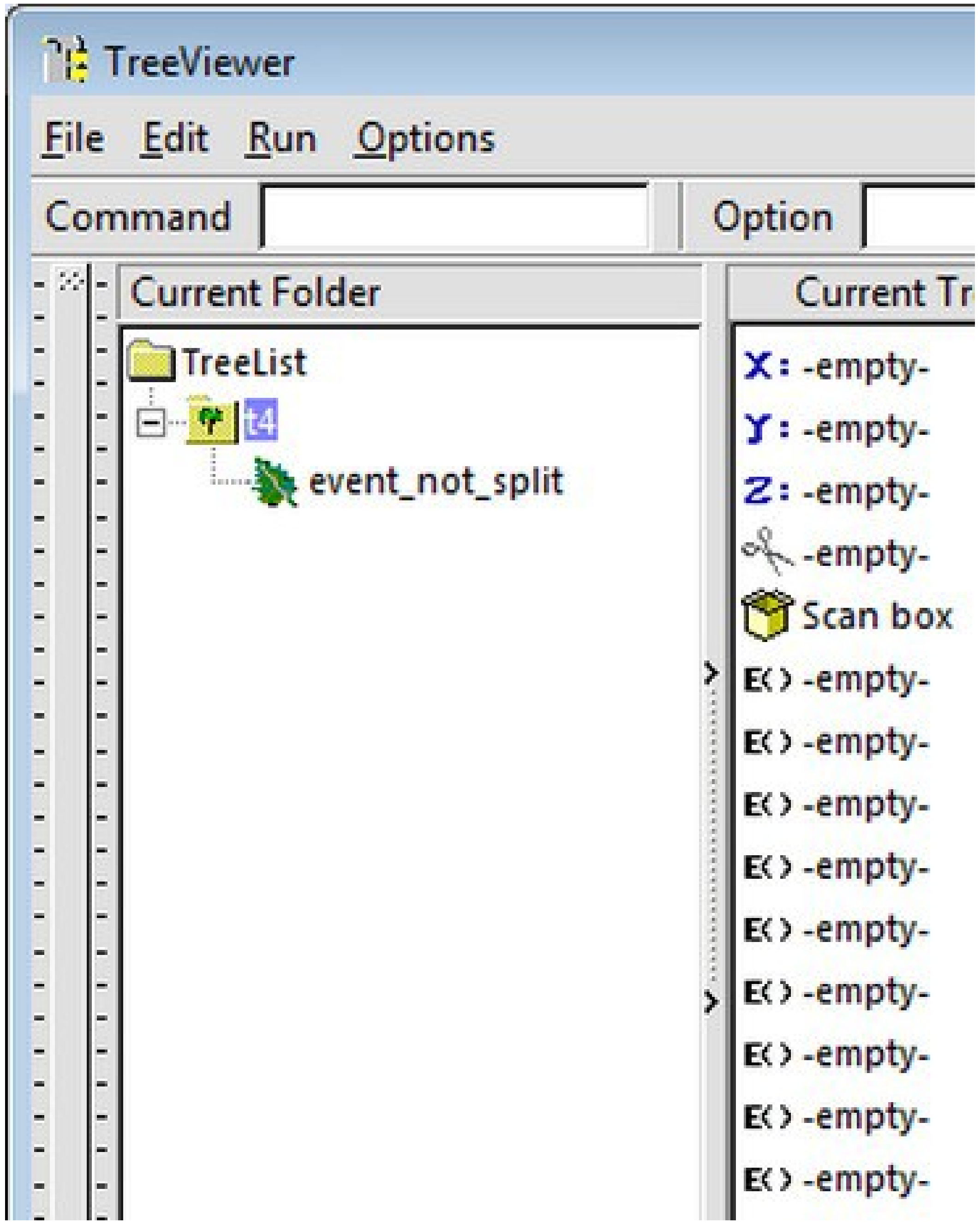
- The third parameter is the address of a pointer to the object to be stored.
- The fourth parameter is the buffer size and is by default 32000 bytes.
- The last parameter is the split-level:

To split a branch means to create a sub-branch for each data member in the object. The split-level can be set to 0 to disable splitting or it can be set to a number between 1 and 99 indicating the depth of splitting.

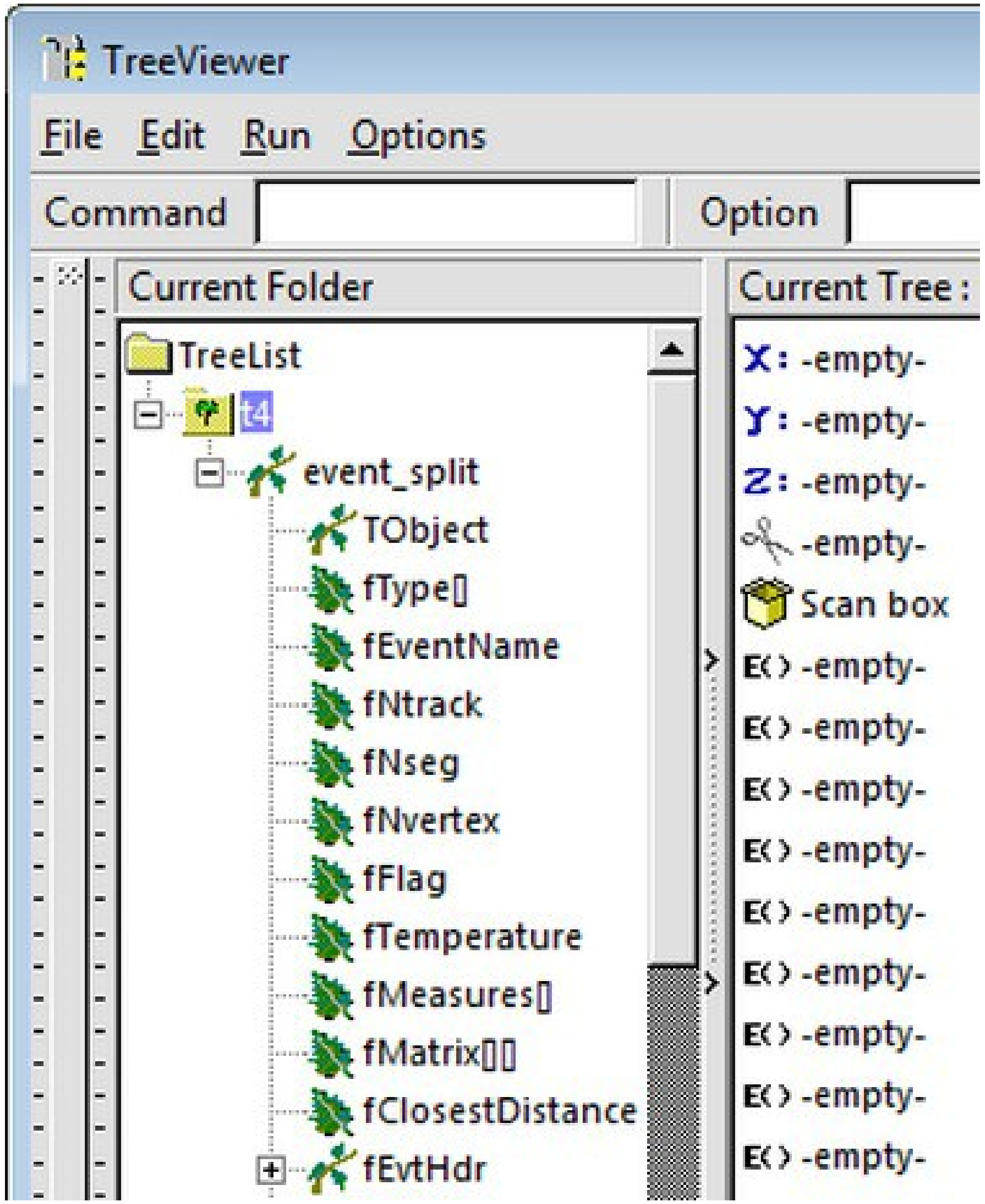
The default for the split-level is 99, → the object will be split to the maximum.

# Splitting

*$ROOTSYS/tutorials/tree/*Tree4.C




Split level = 0

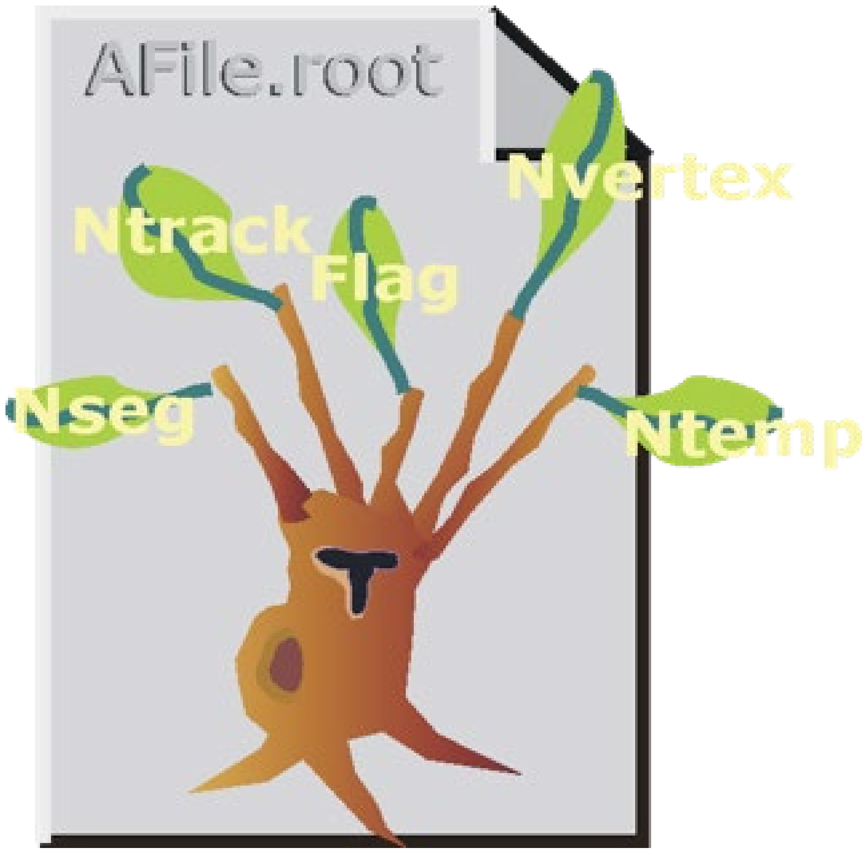



Split level = 99

# Splitting

Setting the split level (default = 99)


Split level = 0




Split level = 99

- Creates one branch per member – recursively
- Allows to browse objects that are stored in trees, even without their library
- Fine grained branches allow fine-grained I/O - read only members that are needed

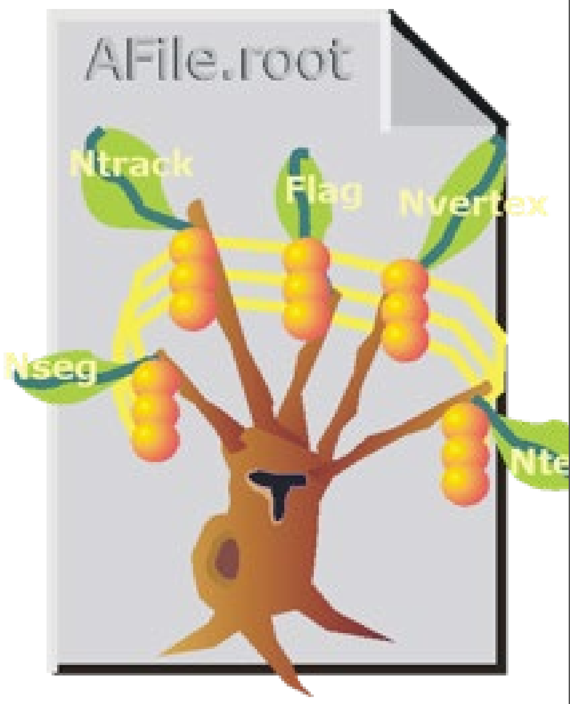
# Performance Considerations

A split branch is:

- Faster to read – if you only want a subset of data members
- Slower to write due to the large number of branches

# Fill the Tree

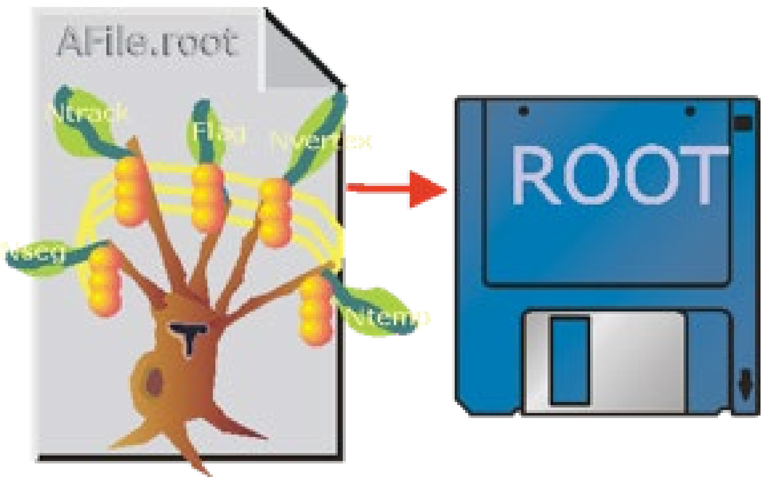
- Loop on the tree
- assign values to the object we want to store
- call `TTree::Fill()` creates a new entry in the tree:
  - snapshot of values of branches' objects




```
for (int e=0;e<100000;++e) {
   myEvent->Generate(e); // fill event
   myTree->Fill();       // fill the tree
}
```

- After, write Tree to file:

```
myTree->Write();
```

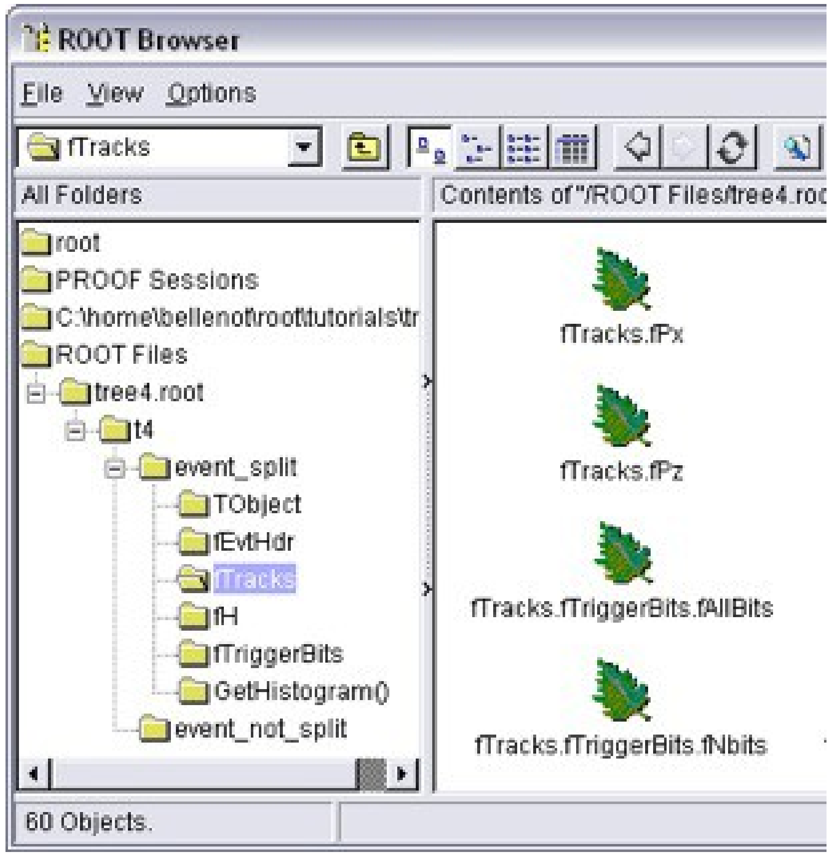


# Reading a Tree

- Open the file and get the TTree object from the file
  - same as we have seen for TNtuple

```
TFile f("AFile.root");
TTree *myTree = 0;
f.GetObject("myTree",myTree);
```

- Or browse the TTree using the TBrowser




- TTree::Print() shows the data layout

# Examples of Writing & Reading Trees

examples: *$ROOTSYS/tutorials/tree*

tree1.C:a tree with several simple (integers and floating point) variables.

tree2.C:a tree built from a C structure

tree3.C: how to extend a tree with a branch from another tree with the Friends feature

tree4.C: a tree with a class (Event)

要使用 Event 这个类，需要到*$ROOTSYS/*test目录make 出 libEvent.so，以便调用。

```
// These examples can be run in many different ways:
// way1: .x tree1.C using the CINT interpreter
// way2: .x tree1.C++ using the automatic compiler interface
// way3: .L tree1.C or .L tree1.C++, then execute functions
```

# How To Read a Tree

- Create a variable pointing to the data

```
Event * myEvent = 0;
```

- Associate a branch with the variable

```
myTree->SetBranchAddress("eBranch",&myEvent);
```

- Read ith-entry in the Tree

```
myTree->GetEntry(i);
```

```
myEvent->GetTracks()->First()->Dump();
==> Dumping object at: 0x0763aad0, name=Track, class=Track
fPx 0.651241 X component of the momentum
fPy 1.02466  Y component of the momentum
fPz 1.2141   Z component of the momentum
[…]
```

# How To Read a Tree

- Example macro

```
void ReadTree() {
  TFile f("AFile.root");
  TTree *T = (TTree*)f->Get("T");
  Event *myEvent = 0;
  TBranch* brEvent = 0;
  T->SetBranchAddress("EvBranch", &myEvent, brEvent);
  T->SetCacheSize(10000000);
  T->AddBranchToCache("EvBranch");
  Long64_t nent = T->GetEntries();
  for (Long64_t i = 0; i < nbent; ++i) {
    brEvent->GetEntry(i);
    myEvent->Analyze();
  }
}
```

Data pointers (e.g. myEvent) MUST be set to 0

# Accessing Tree Branches

- If we are interested in only some branches of a Tree:
  - Use `TTree::SetBranchStatus()` or `TBranch::GetEntry()` to select the branches to be read
    - by defult all branches are read when calling `TTree::GetEntry(event_number)`
  - Speed up considerably the reading phase
  - Example: we are interested in reading only a branch with an array of muons

```
TClonesArray* myMuons = 0;
// disable all branches
myTree->SetBranchStatus("*", 0);
// re-enable the "muon" branches
myTree->SetBranchStatus("muon*", 1);
myTree->SetBranchAddress("muon", &myMuons);
// now read (access) only the "muon" branches
for (Long64_t i = 0; i < myTree->GetEntries(); ++i) {
    myTree->GetEntry(i);
```

# Copy subset of Tree to new Tree

```
void copytree() {
   gSystem->Load("$ROOTSYS/test/libEvent");
   //Get old file, old tree and set top branch address
   TFile *oldfile = new TFile("$ROOTSYS/test/Event.root");
   TTree *oldtree = (TTree*)oldfile->Get("T");
   Event *event   = new Event();
   oldtree->SetBranchAddress("event",&event);
   oldtree->SetBranchStatus("*",0);
   oldtree->SetBranchStatus("event",1);
   oldtree->SetBranchStatus("fNtrack",1);
   oldtree->SetBranchStatus("fNseg",1);
   oldtree->SetBranchStatus("fH",1);

   //Create a new file + a clone of old tree in new file
   TFile *newfile = new TFile("small.root","recreate");
   TTree *newtree = oldtree->CloneTree();

   newtree->Print();
   newfile->Write();
   delete oldfile;
   delete newfile;
}
```

参考： tutorials/tree/copytree.C

# Tree Selection Syntax

- Syntax for querying a tree
  - Print the first 8 variables of the tree:

```
MyTree->Scan();
```

  - Prints all the variables of the tree:

```
MyTree->Scan("*");
```

  - Prints the values of var1, var2 and var3.

```
MyTree->Scan("var1:var2:var3");
```

  - A selection can be applied in the second argument:
  - Prints the values of var1, var2 and var3 for the entries where var1 is greater than 0

```
MyTree->Scan("var1:var2:var3", "var1>0");
```

- Use the same syntax for `TTree::Draw()`

# Looking at the Tree

- TTree::Scan("leaf:leaf:….") shows the values

```
root [] myTree->Scan("fNseg:fNtrack"); > scan.txt

root [] myTree->Scan("fEvtHdr.fDate:fNtrack:fPx:fPy","",
               "colsize=13 precision=3 col=13:7::15.10");
```

```
*******************************************************************************
* Row * Instance * fEvtHdr.fDate * fNtrack *           fPx *            fPy *
*******************************************************************************
*   0 *        0 *        960312 *     594 *          2.07 *     1.459911346 *
*   0 *        1 *        960312 *     594 *         0.903 *   -0.4093382061 *
*   0 *        2 *        960312 *     594 *         0.696 *    0.3913401663 *
*   0 *        3 *        960312 *     594 *        -0.638 *     1.244356871 *
*   0 *        4 *        960312 *     594 *        -0.556 *   -0.7361358404 *
*   0 *        5 *        960312 *     594 *         -1.57 *   -0.3049036264 *
*   0 *        6 *        960312 *     594 *        0.0425 *    -1.006743073 *
*   0 *        7 *        960312 *     594 *          -0.6 *    -1.895804524 *
```

# Looking at the Tree

- TTree::Print() shows the data layout

```
root [] TFile f("AFile.root")
root [] myTree->Print();
******************************************************************************
*Tree    :myTree    : A ROOT tree                                            *
*Entries :       10 : Total =          867935 bytes  File  Size =     390138 *
*        :          : Tree compression factor =   2.72                       *
******************************************************************************
*Branch  :eBranch                                                        *
*Entries :       10 : BranchElement (see below)                              *
*............................................................................*
*Br    0 :fUniqueID :                                                        *
*Entries :       10 : Total  Size=         698 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=       64000 bytes  Compression=   1.00     *
*............................................................................*
…
…
```

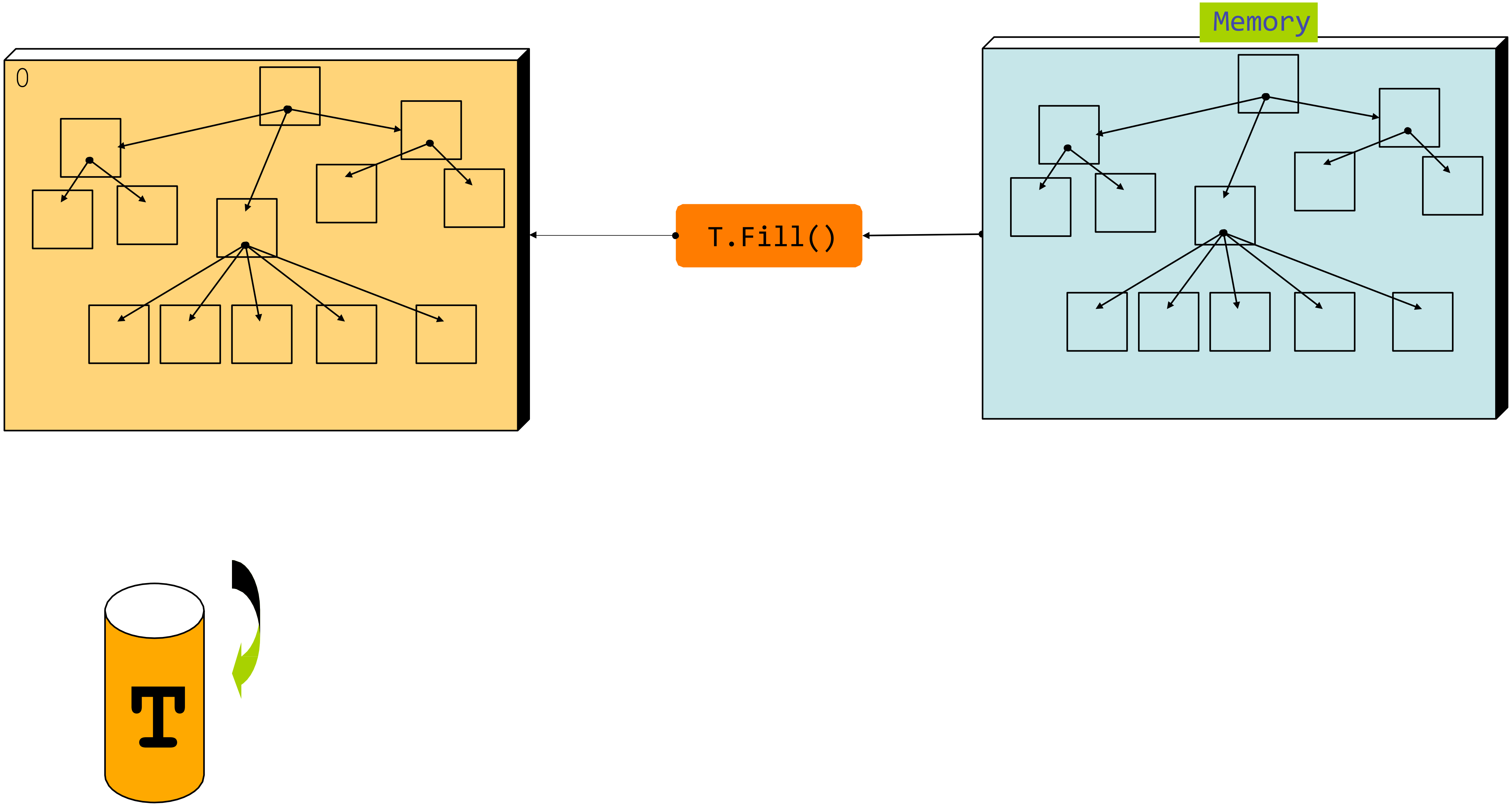
# Looking at the Tree

- `TTree::Show(entry_number)` shows values for one entry

```
root [] myTree->Show(0);
======> EVENT:0
eBranch            = NULL
fUniqueID          = 0
fBits              = 50331648
[...]
fNtrack            = 594
fNseg              = 5964
[...]
fEvtHdr.fRun       = 200
[...]
fTracks.fPx        = 2.066806, 0.903484, 0.695610,-0.637773,…
fTracks.fPy        = 1.459911, -0.409338, 0.391340, 1.244357,…
```

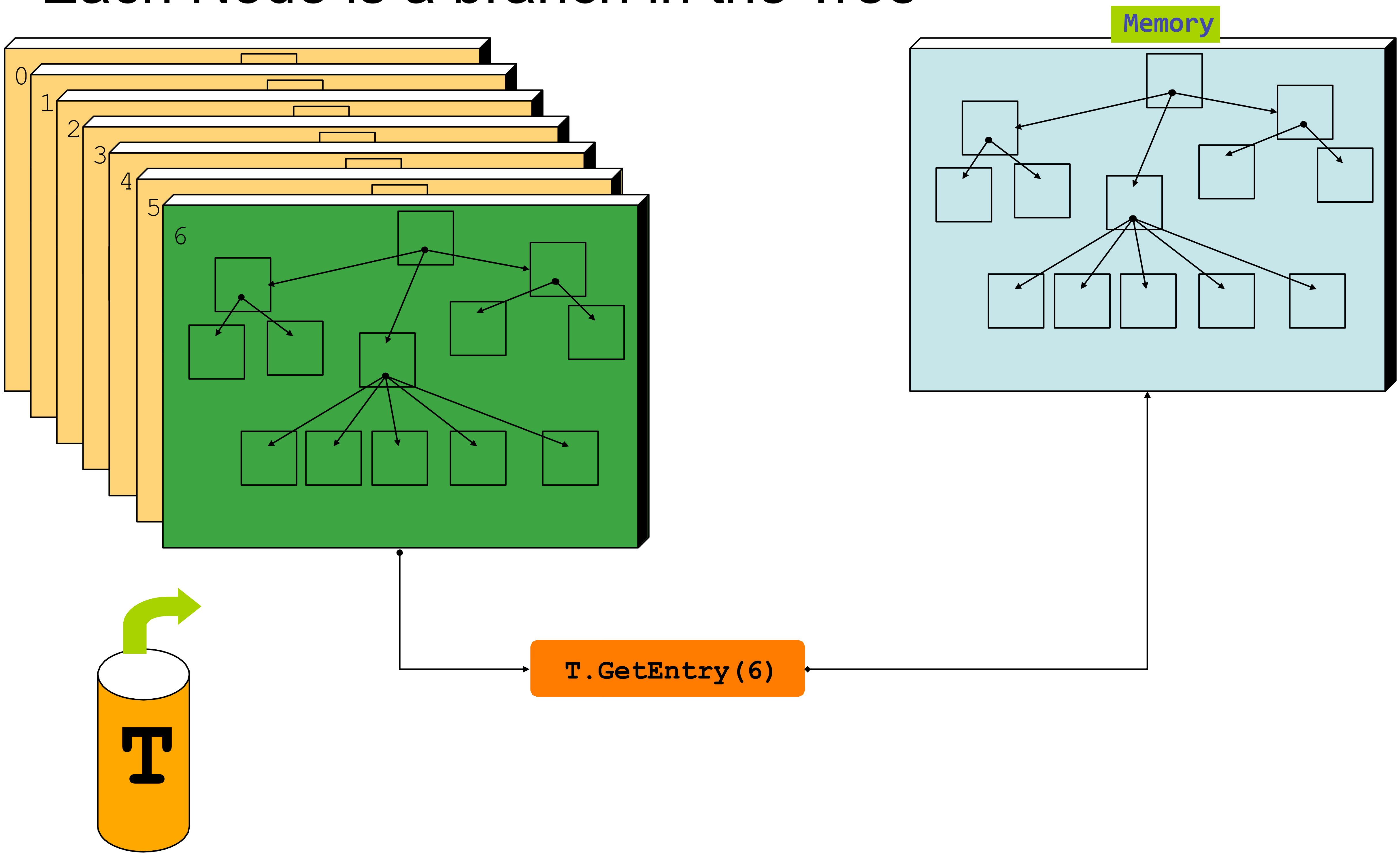
# Memory ↔ Tree

- Each Node is a branch in the Tree



# Memory ↔ Tree

- Each Node is a branch in the Tree



# TChain: The Forest

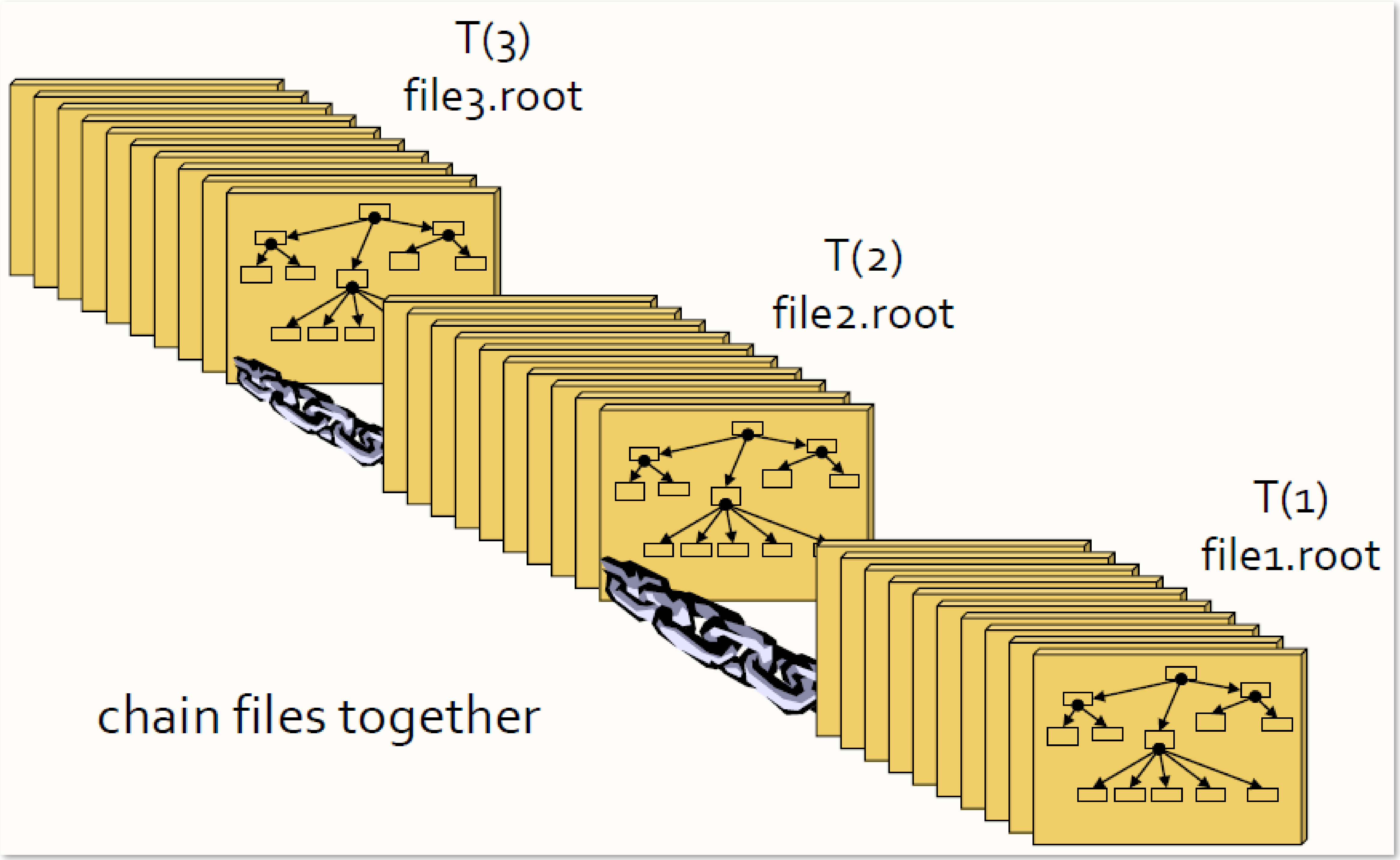
- Collection of Trees:
  - list of ROOT files containing the same tree
- Same semantics as `TTree`.
  - As an example, assume we have three files called file1.root, file2.root, file3.root. Each contains tree called "T". Create a chain:

```
TChain chain("T"); // argument: tree name
chain.Add("file1.root");
chain.Add("file2.root");
chain.Add("file3.root");
```

  - Now we can use the `TChain` like a `TTree!`

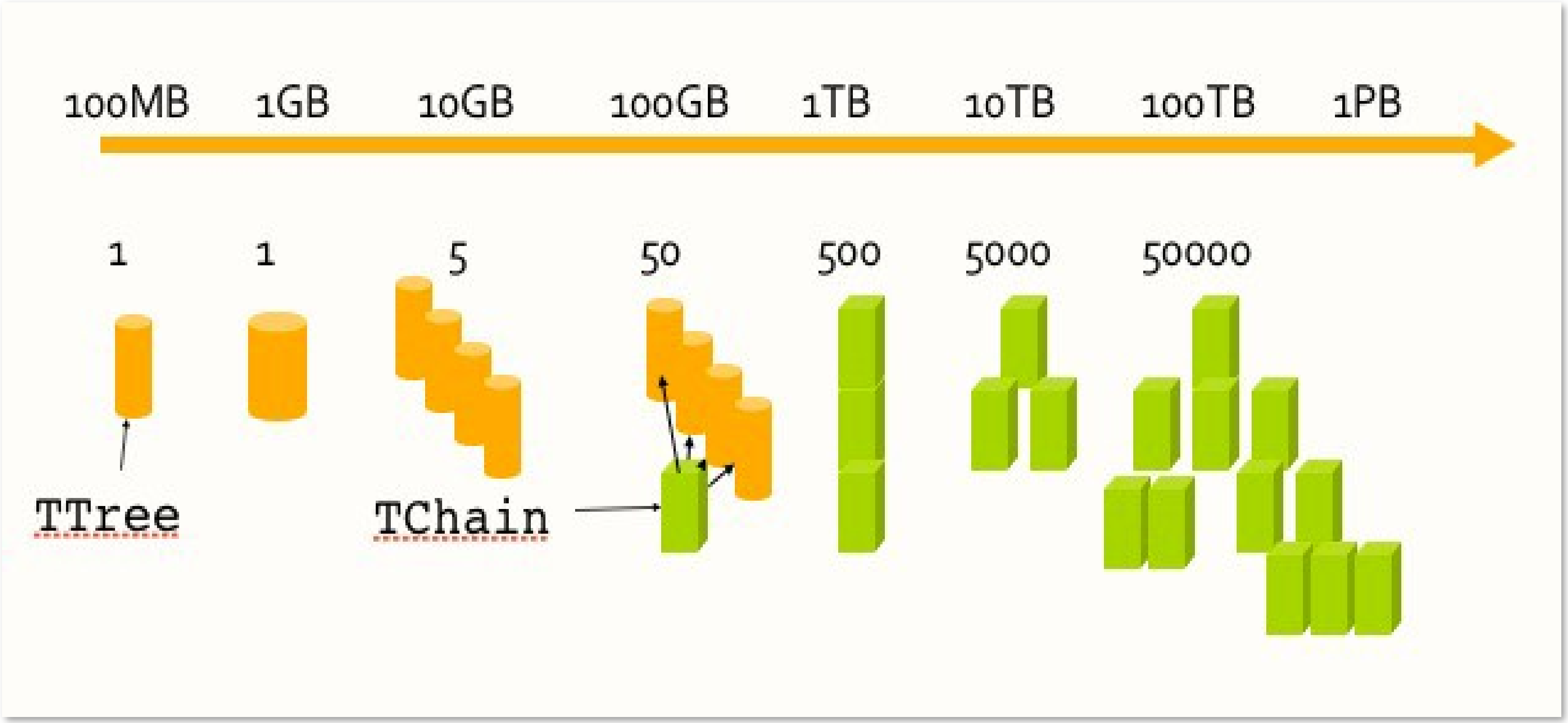
# TChain

- Chain Files together



# Data Volume and Organization

- A `TFile` typically contains 1 `TTree`
- A `TChain` is a collection of `TTrees` or/and `TChains`

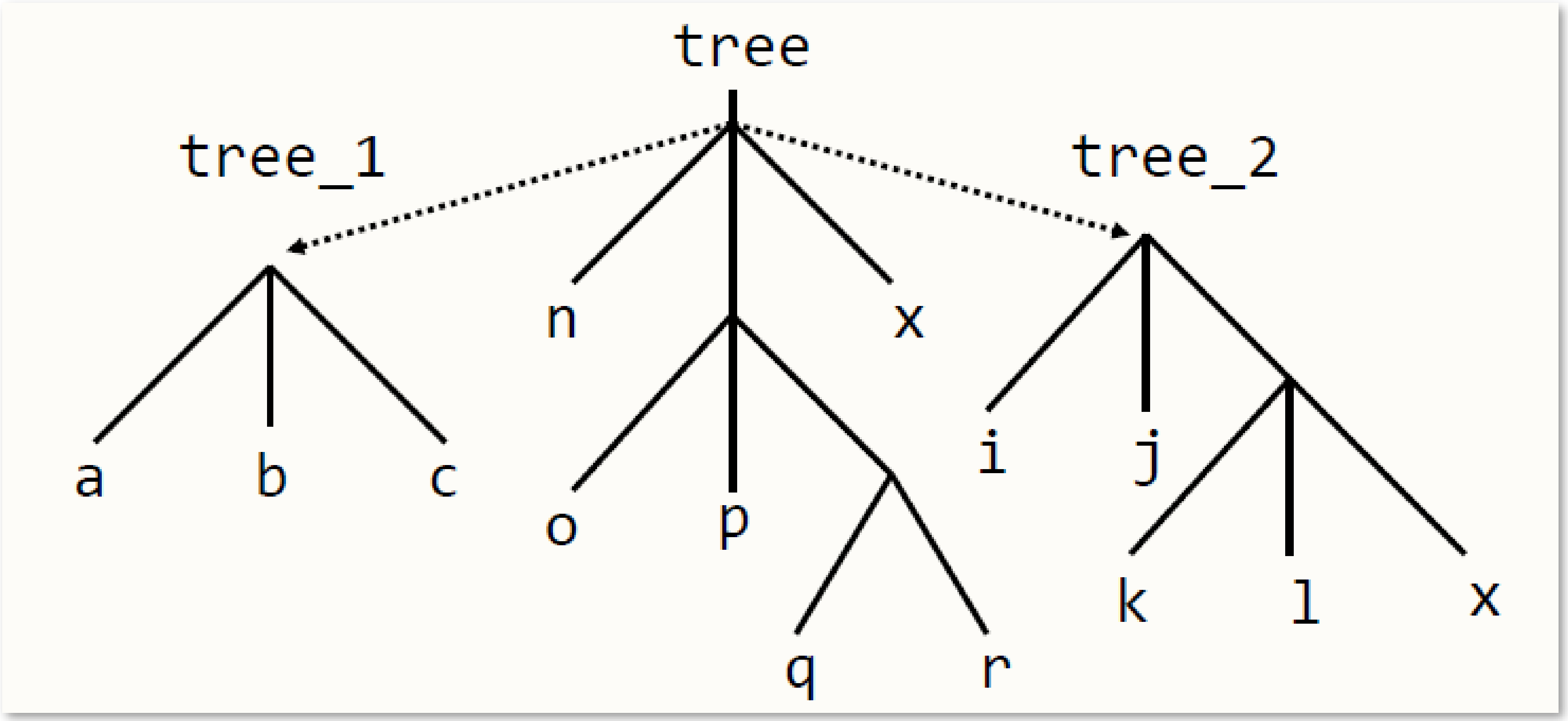


# Adding Friends to Trees

- Trees are designed to be read only
- Often, people want to add branches to existing trees and write their data into it
- Using tree friends is the solution:
  - Create a new file holding the new tree
  - Create a new Tree holding the branches for the user data
  - Fill the tree/branches with user data
  - Add this new file/tree as friend of the original tree

# Tree Friends

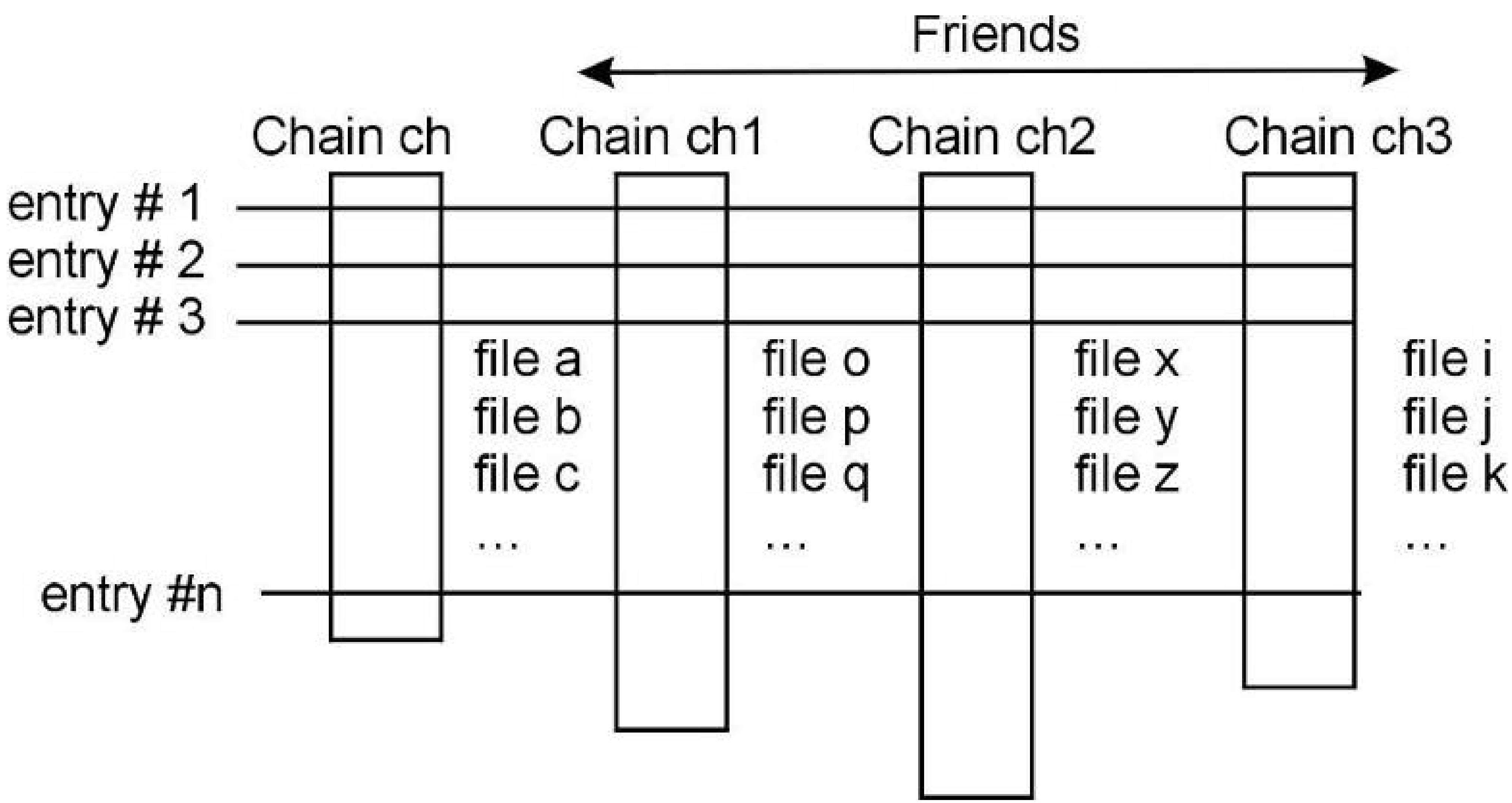
- Using Tree Friends




```
TFile f1("tree.root");
tree.AddFriend("tree_1", "tree1.root")
tree.AddFriend("tree_2", "tree2.root");
tree.Draw("x:a", "k<c");
tree.Draw("x:tree_2.x");
```

# TTree::AddFriend

The number of entries in the friend must be equal or greater to the number of entries of the original chain. If the friend has fewer entries a warning is given and the resulting histogram will have missing entries.




A full example of a tree and friends is in Example #3 ($ROOTSYS/tutorials/tree/tree3.C)

# Trees in Analysis

- The methods **TTree**::Draw, **TTree**::MakeClass and **TTree**::MakeSelector are available for data analysis.
  - The **TTree::Draw** method is a powerful yet simple way to look and draw the trees contents. It enables you to plot a variable (a leaf) with just one line of code.
  - The **TTree::MakeClass** creates a class that loops over the trees entries one by one. You can then expand it to do the logic of your analysis.
  - The **TTree::MakeSelector** is the recommended method for ROOT data analysis, especially important for large data set in a parallel processing configuration

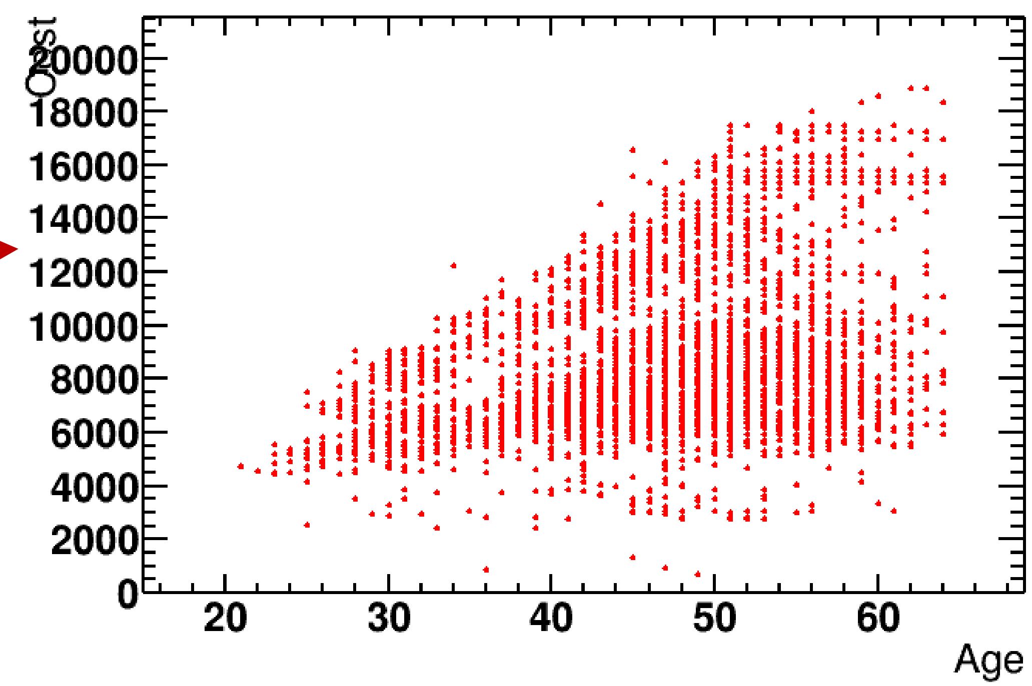
# TTree:Draw()

```
Long64_t Draw(const char* varexp, const char* selection,
              Option_t* option = "",
              Long64_t nentries = 1000000000,
              Long64_t firstentry = 0)
```

option is same as **TH1::Draw** method.

We use the tree in cernstaff.root, made by $ROOTSYS/tutorials/tree/cernbuild.C

```
//root file from tree/staff.C:
TFile f ("cernstaff.root");
T->Draw("Cost") //1-D plot
T->Draw("Cost:Age") //2-D plot
T->Draw("Cost:Age:Children") //3-D plot
```




## Using selection with TTree:Draw :

selection = "weight *(boolean expression)"

The value of the selection is used as a weight when filling the histogram: **any C++ operator, and some functions defined in TFormula can be used**

# Using TCut Objects in Ttree::Draw

A **TCut** is a specialized string object used for TTree selections.

```
TCut cut1 = "x<1"
TCut cut2 = "y>2"
//then cut1 && cut2
//result is the string "(x<1)&&(y>2)"
```

Operators " =, +=, +, *, !, &&, || " are overloaded

```
root[] TCut c1 = "x < 1"
root[] TCut c2 = "y < 0"
root[] TCut c3 = c1 && c2
root[] MyTree.Draw("x", c1)
root[] MyTree.Draw("x", c1 || "x>0")
root[] MyTree.Draw("x", c1 && c2)
root[] MyTree.Draw("x", "(x + y)*(c1 && c2)")
```

# Setting the Range in TTree::Draw

Ttree::Draw() has two more optional parameters:

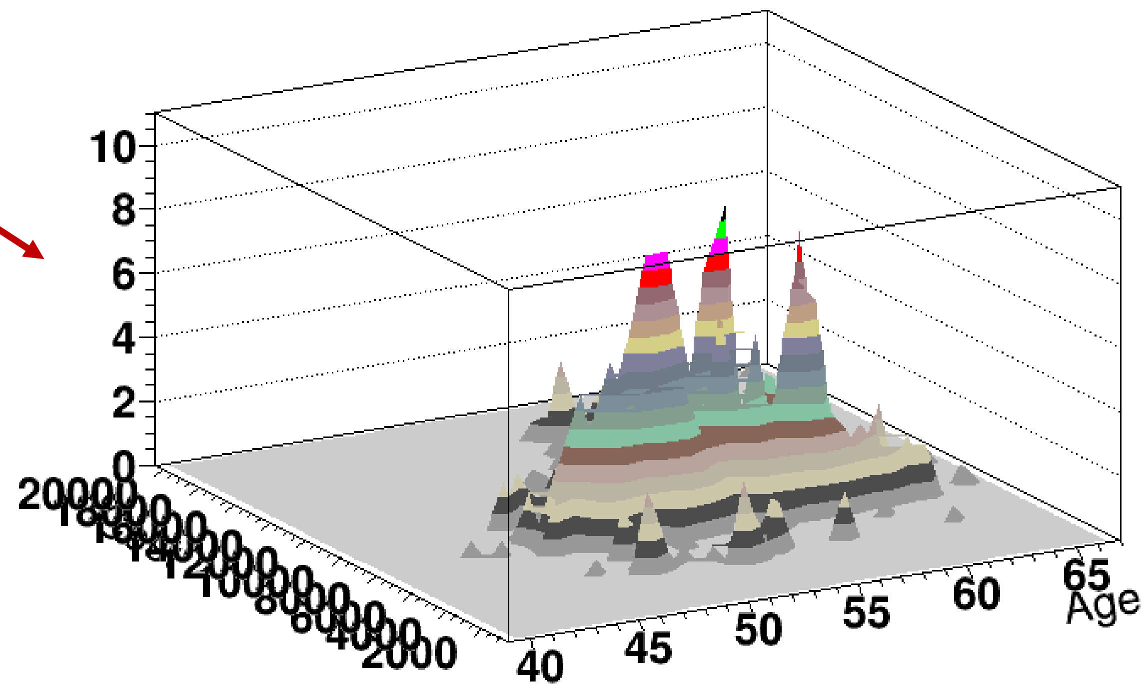
one is the number of entries and the second one is the entry to start with.

For example, this command draws 1000 entries starting with entry 100:

```
T->Draw("Cost:Age", "Nation == \"FR\"","surf2",1000,100);
```

**TTree::Draw** Examples:

exercise and understand the draw commands in section 12.20.7-12.20.7.1 of User guide.




https://root.cern.ch/root/htmldoc/guides/users-guide/ROOTUsersGuide.html#simple-analysis-using-ttreedraw

# Using TTree::Scan

TTree::Scan can be used to print the content of the tree's entries:

```
Long64_t Scan(const char* varexp = "", const char*
            selection = "", Option_t* option = "", Long64_t
            nentries = 1000000000, Long64_t firstentry = 0)
```

```
root[] MyTree->Scan();// print the first 8 variables of the tree.
root[] MyTree->Scan("*"); //print all the variable of the tree.
//Specific variables of the tree
root[] MyTree->Scan("var1:var2:var3");
//A selection can be applied in the second argument
root[] MyTree->Scan("var1:var2:var3","var1==0");
```

**TTree**::Scan returns the number of entries passing the selection.
By default 50 rows are shown before **TTree::Scan** pauses.

To change the default number of rows, use **TTree::SetScanfield(maxrows).**

If 0 is set, all rows are shown. →

```
root[] tree->SetScanField(0);
```

This option is interesting when dumping the contents of a Tree to an ascii file, e.g. from the command line: →

```
root[] tree->Scan("*");>log.txt
```

# Filling a Histogram

The TTree::Draw method can also be used to fill a specific histogram:

```
root[] TFile *f = new TFile("Event.root")
root[] T->Draw("fNtrack >> myHisto")
root[] myHisto->Print()
TH1.Print Name= myHisto, Entries= 100, Total sum= 100
//to append more entries to the histogram
root[] T->Draw("fNtrack >>+ myHisto")
```

```
//To set the number of bins for a specific histogram
tree.Draw("sqrt(x)>>hsqrt(500,10,20)");
// plot sqrt(x) between 10 and 20 using 500 bins
tree.Draw("sqrt(x):sin(y)>>hsqrt(100,10,,50,.1,.5)");
// plot sqrt(x) against sin(y) 100 bins in x-direction; lower
// limit on x-axis is 10; no upper limit; 50 bins in y-direction;
// lower limit on y-axis is .1; upper limit is .5
tree.Draw("sqrt(x)>>+hsqrt","y>0");
//will not reset hsqrt and continue filling the histogram
```

appending the histogram with a “+”, will not reset hsqrt, but will continue to fill it.

# Tree Information

Once we have drawn a tree, we can get information about the tree.

**GetV1**: Returns a pointer to the float array of the first variable.

**GetV2**: Returns a pointer to the float array of second variable

**GetV3**: Returns a pointer to the float array of third variable.

**GetW**: Returns a pointer to the float array of Weights where the weight equals the result of the selection expression.

```
root[] TFile *f = new TFile("Event.root")
root[] T->Draw("fNtrack")
root[] Float_t *a
root[] a = T->GetV1()
//Loop through the first 10 entries and print the values of fNtrack:
root[] for (int i = 0; i < 10; i++)
root[] cout << a[i] << " " << endl
// need an endl to see the values
594 597 606 595 604 610 604 602 603 596
```

# Analyzing Trees

- Tree is an efficient storage and access for huge amounts of structured data
- Allows selective access of data
- It is used to analyze and select data.
- Most convenient way to analyze data store in a Tree is with the **`TSelector`** class
  - the user creates a new class `MySelector` deriving from **`TSelector`**
  - the `MySelector` object is used in `TTree::Process(TSelector*,...)`
  - ROOT invokes the `TSelector'`s functions which are virtuals, so the user provided function implemented in `MySelector` will be called.

**Almost all HEP analyses based on TTree**

# 举例 用脚本读取MC truth 信息

**1. 一个显示root文件中MC衰变信息的程序：**

```
> cat stag.C
#include <iostream>
#include "TFile.h"
#include "TChain.h"
#include "TTree.h"
using namespace std;
int main() {
  TChain * chain = new TChain("truth");
  chain->Add("in.root");
  Int_t           indexmc;
  Int_t           pdgid[100];
  Int_t           motheridx[100];
  chain->SetBranchAddress("indexmc",&indexmc);
  chain->SetBranchAddress("pdgid",pdgid);
  chain->SetBranchAddress("motheridx",motheridx);
  for (Long64_t i=0; i<5;i++) {
    chain->GetEntry(i);
    cout << "event = " << i << endl;
    for ( int k = 0; k != indexmc; k++ ) {
      cout << "  " << k << " id= " << pdgid[k]
           << " mo=" << motheridx[k] << " "
           << pdgid[motheridx[k]] << endl;
  }
}
```

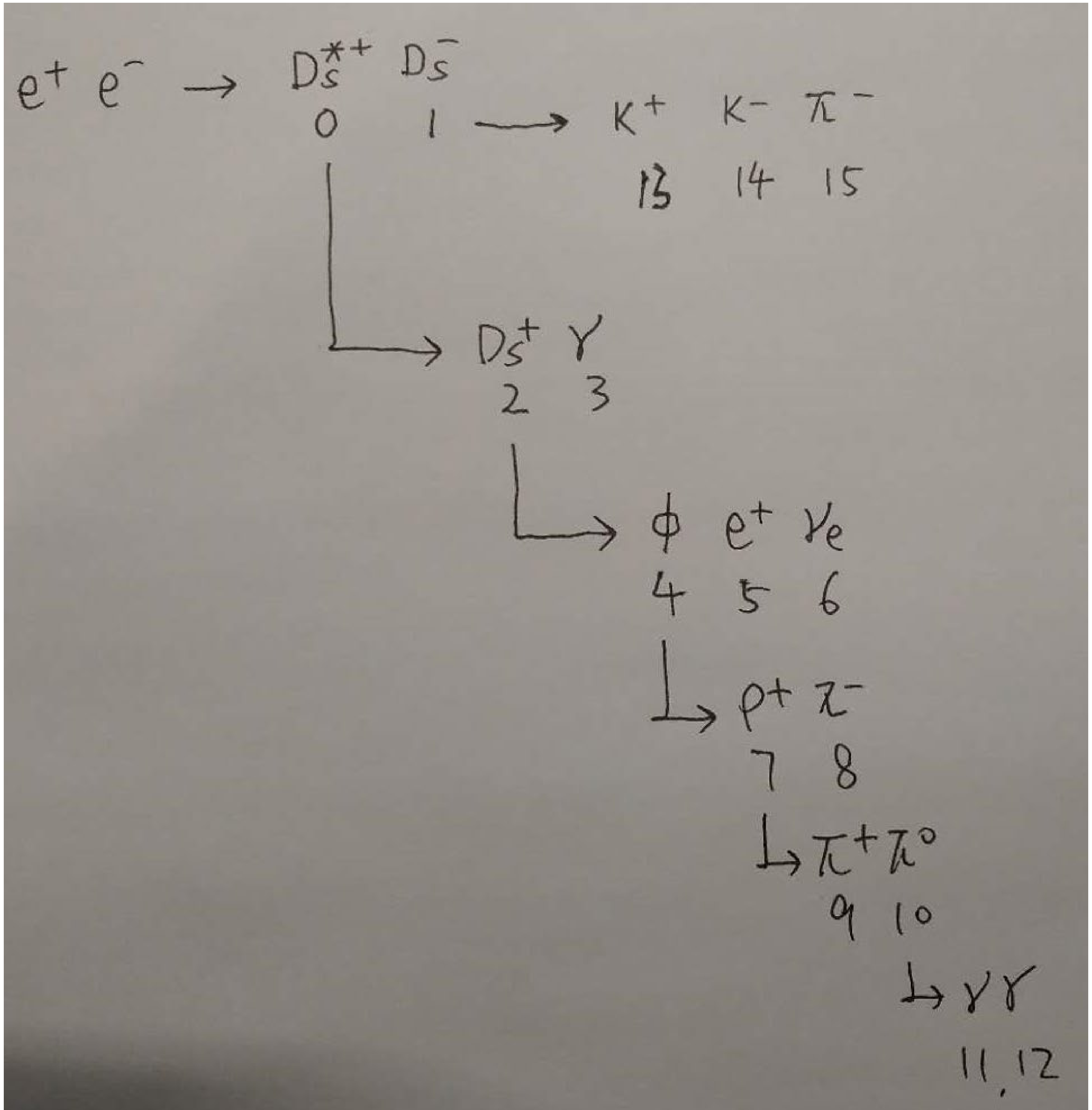
**2. 编译成可执行文件：**

```
>g++ stag.C `root-config --libs`
  `root-config --cflags` -o run
```

**3. 运行程序：**

```
> ./run
event = 0
   0  id=   433 mo=0    433
   1  id=  -431 mo=1   -431
   2  id=   431 mo=0    433
   3  id=    22 mo=0    433
   4  id=   333 mo=2    431
   5  id=   -11 mo=2    431
   6  id=    12 mo=2    431
   7  id=   213 mo=4    333
   8  id=  -211 mo=4    333
   9  id=   211 mo=7    213
   10 id=   111 mo=7    213
   11 id=    22 mo=10   111
   12 id=    22 mo=10   111
   13 id=   321 mo=1   -431
   14 id=  -321 mo=1   -431
   15 id=  -211 mo=1   -431
Event = 1
…… 以下省略
```

# 举例 用脚本读取MC truth 信息




```
> ./run
event = 0
   0  id=   433  mo=0    433
   1  id=  -431  mo=1   -431
   2  id=   431  mo=0    433
   3  id=    22  mo=0    433
   4  id=   333  mo=2    431
   5  id=   -11  mo=2    431
   6  id=    12  mo=2    431
   7  id=   213  mo=4    333
   8  id=  -211  mo=4    333
   9  id=   211  mo=7    213
   10 id=   111  mo=7    213
   11 id=    22  mo=10   111
   12 id=    22  mo=10   111
   13 id=   321  mo=1   -431
   14 id=  -321  mo=1   -431
   15 id=  -211  mo=1   -431
Event = 1
......  以下省略
```

# Example Macro

一个实际工作的例子truth.C

```
#include "TFile.h"
#include "TChain.h"
#include "TTree.h "
int main() {
   TChain * chain = new TChain("truth");
   chain->Add("in-*.root");
   Int_t indexmc;
   Int_t pdgid[100];
   Int_t motheridx[100];
   chain->SetBranchAddress("indexmc", &indexmc);
   chain->SetBranchAddress("pdgid", pdgid);
   chain->SetBranchAddress("motheridx", motheridx);
   TFile *file = new TFile("out.root", "recreate");
   Int_t nt_Dspmode, nt_Dsmmode, Dspmode, Dsmmode;
   tree->Branch("Dspmode",  &nt_Dspmode,  "nt_Dspmode/I");
   tree->Branch("Dsmmode",  &nt_Dsmmode,  "nt_Dsmmode/I");
   Long64_t nentries = chain->GetEntries();
   for (Long64_t i=0; i<nentries;i++) {
       chain->GetEntry(i);
       TagModeMatch(Dspmode, Dsmmode, indexmc, pdgid, motheridx);
       nt_Dspmode = Dspmode;
       nt_Dsmmode = Dsmmode;
       tree->Fill();
   }
   file->Write();
}
```

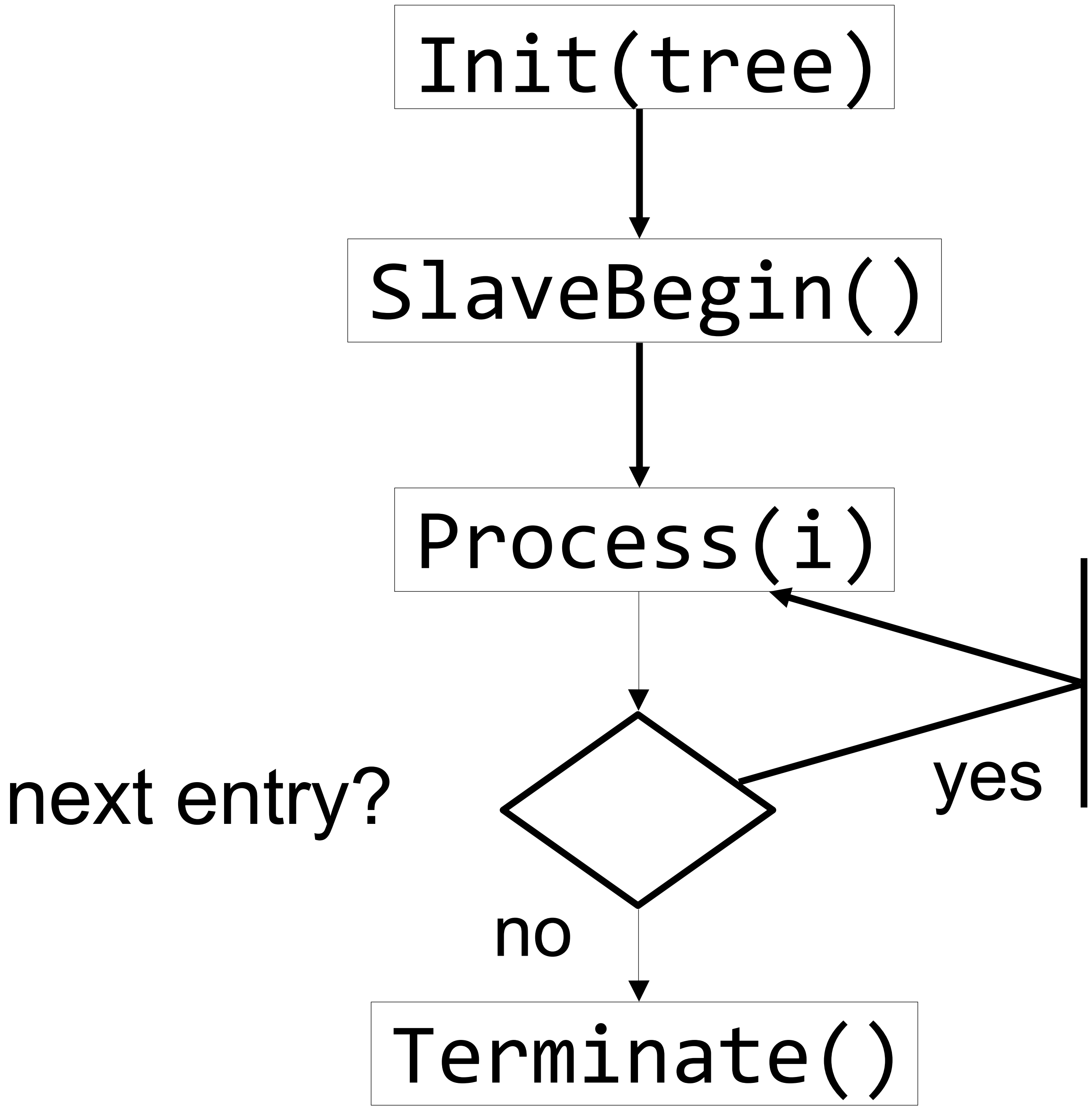
加入所有以in-开头，扩展名为.root的文件

根据indexmc, pdgid, motheridx 判断 Ds+ 和Ds-的衰变模式

编译：g++ truth.C `root-config --libs` `root-config --cflags` -o run_truth

# Tree Data Access

E.g. `tree->Process("MySelector.C+")`



# TSelector

Steps of ROOT using a `TSelector`:

1. ***setup*** **`TMySelector::Init(TTree *tree)`**
   `fChain = tree; fChain->SetBranchAddress()`

2. ***start*** **`TMySelector::SlaveBegin()`**
   create histograms

3. ***run*** **`TMySelector::Process(Long64_t)`**
   `fChain->GetTree()>GetEntry(entry);`
   analyze data, fill histograms,…

4. ***end*** **`TMySelector::Terminate()`**
   fit histograms, write them to files,…

# Using TTree::MakeClass

When you need to do some programming with the variable in the tree, use **TTree::MakeClass**

```
root[] .L libEvent.so // in $ROOTSYS/test
root[] TFile *f = new TFile("Event.root");
root[] f->ls();
TFile**Event.rootTTree benchmark ROOT file
 TFile*Event.rootTTree benchmark ROOT file
KEY: TH1Fhtime;1 Real-Time to write versus time
KEY: TTree T;1 An example of a ROOT tree
root[] T->MakeClass("MyClass")
Files: MyClass.h and MyClass.C generated from Tree: T
```

MyClass.h contains the class definition,

MyClass.C contains the **MyClass::Loop()** method.

Modify **MyClass::Loop** to implement analysis: select entries, fill histograms, draw plots and output files.

Load MyClass and execute the **Loop()** function

# Using TTree::MakeSelector

- With a TTree to make a selector to process a limited set of entries:

especially important in a parallel processing configuration where we can specify which entries to send to a processor.

the **TTree::MakeSelector** method creates two files similar to TTree::MakeClass

The **TTree::Process** method is used to specify the selector and the entries

- In the resulting files is a class that is a descendent of TSelector and implements the following methods:

**TSelector::Begin()** - is called every time a loop over the tree starts.

This is a convenient place to create your histograms.

**TSelector::Process()** - is called to process an event.

It is the user's responsibility to read the TTree entries in memory,

apply entry selections and fill the histograms.

**TSelector::Terminate()** - is called at the end of a loop on a **TTree**.

This is a convenient place to draw and fit your histograms.

# Tree analysis Example: h1analysis.C

Example of analysis class for the H1 data:
(参考：[https://root.cern.ch/doc/master/h1analysis_8C.html](https://root.cern.ch/doc/master/h1analysis_8C.html))

h1analysis.C uses 4 large data sets from the H1 collaboration.
(需要下载 [https://root.cern.ch/download/h1analysis/](https://root.cern.ch/download/h1analysis/) 的4个数据文件，共200多MB)

A chain of 4 files is used to illustrate the various ways to loop on Root data sets.
Each data set contains a Root Tree named "h42" .

The class definition in h1analysis.h has been generated automatically by TTree::MakeSelector using one of the files with the following statement:

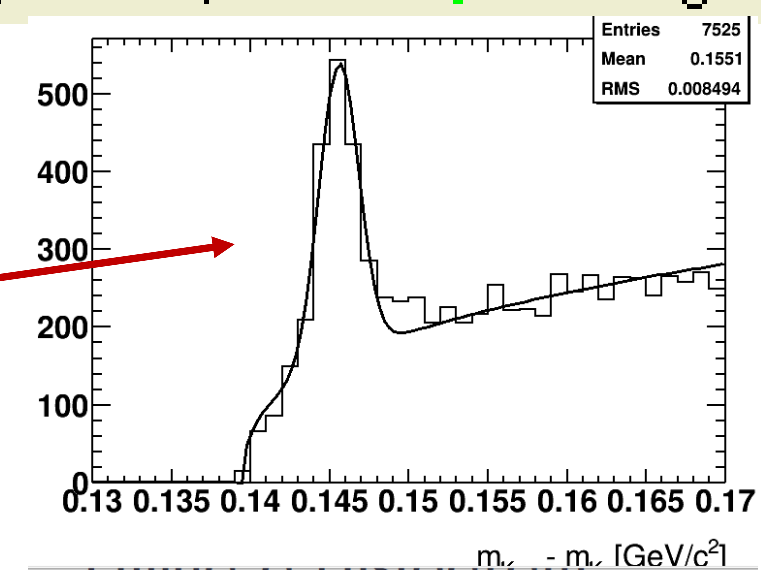
```
root[] h42>MakeSelector("h1analysis");
```

```
root [0] TChain chain("h42");
root [1] chain.Add("dstarmb.root");
root [2] chain.Add("dstarp1a.root");
root [3] chain.Add("dstarp1b.root");
root [4] chain.Add("dstarp2.root");
root [5] chain.Process("h1analysis.C")
```

This produces two files: h1analysis.h and h1analysis.C (skeleton of this file).
The h1analysis class is derived from the Root class TSelector.

Loop on all events:

```
root[] chain.Process("h1analysis.C")
```



# Summary

- The ROOT Tree is one of the most powerful collections available for HEP
- Extremely efficient for huge number of data sets with identical layout
- Very easy to look at `TTree` - use `TBrowser`!
- Write once, read many: ideal for experiments' data; use friends to extend
- Branches allow granular access; use splitting to create branch for each member, even through collections
- `TSelector` class provides a powerful way of processing the Tree data using compiled code