A Short Course on ROOT
Day 3

Subir Sarkar
SINP, Kolkata

VIIIth SERC School on EHEP, VECC, Kolkata
June 20 – July 10, 2011

Course Schedule – Day 3

- Physics analysis using ROOT
  - HEP analysis performed mainly with ROOT
  - there are several ways
    - simple macro to compiled classes
    - TNtuple, Ttree, TChain
    - TClonesArray, TSelector, TCut

TNtuple

A simple tree restricted to a list of float variables only.
Each variable goes to a separate branch

```
void basic() {
   ifstream in;
   in.open("basic.dat",”ios::in”); // is opened successfully?
   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) { // read until the end of the file
     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();
}
```

Trees

- Efficient storage and access for huge amounts of structured data
  - allows selective access of data
  - TTree knows its layout
- Trees allow direct and random access to any e

  - sequential access is the best
- Trees have branches and leaves
  - one can read a subset of all branches
- Optimized for network access (read-ahead)
- High level functions like TTree::Draw loop on all entries with selection expressions
- Trees can be browsed via TBrowser
- Trees can be analyzed via TTreeViewer

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 structure

- Branches: directories
- Leaves: data containers
- Can read a subset of all branches
  - speeds up considerably the data analysis processes
- Branches of the same TTree can be written to separate files

Five Steps to Build a Tree

1. Create a TFile
2. Create a TTree
3. Add TBranch to the TTree
4. Fill the tree
5. Write the file

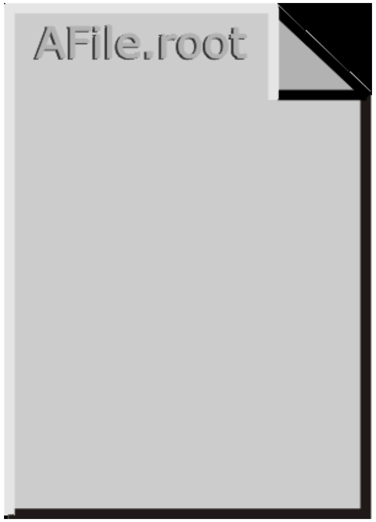
Example code

```
void WriteTree()
{
  TFile f("AFile.root", "RECREATE"); ❶

  TTree *t = new TTree("myTree","A Tree"); ❷

  Event *myEvent = new Event();
  t->Branch("EventBranch", &myEvent);  ❸

  for (int e=0;e<100000;++e) {  ❹
    myEvent->Generate(); // hypothetical
    t->Fill();
  }

  t->Write(); ❺
}
```

Step 1: Create a TFile Object

- Trees can be huge
  - open a file for swapping filled entries
  - file has the ownership

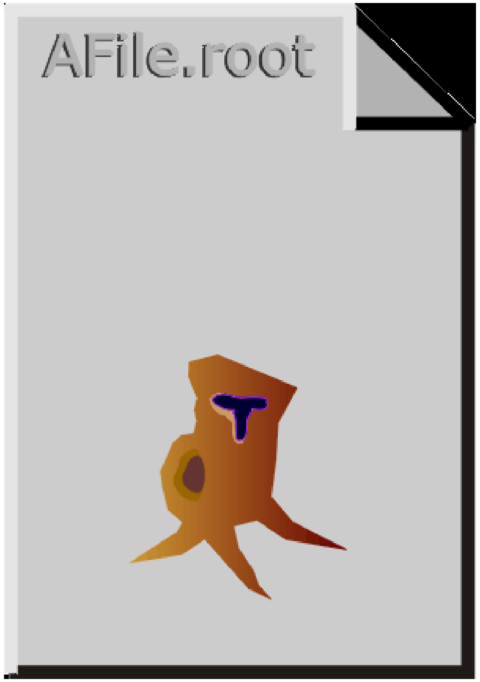
```
TFile *hfile = TFile::Open("AFile.root",
                          "RECREATE");
```



Step 2: Create a TTree Object

The TTree constructor

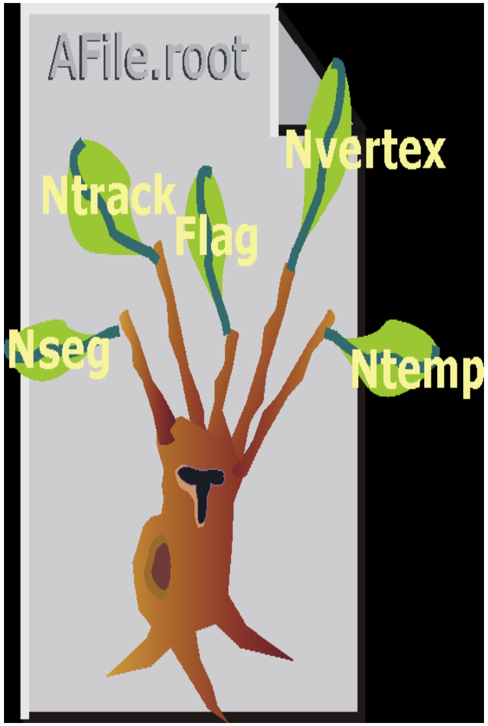
- Tree name (e.g. "myTree")
- Tree title




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

Step 3: Adding a Branch

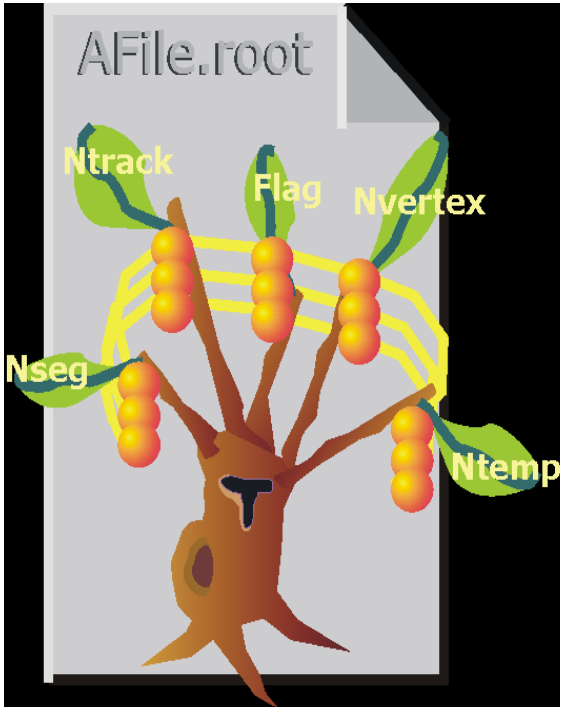
- Branch name
- Address of pointer to the object




```
Event *myEvent = new Event();
myTree->Branch("eBranch",&myEvent);
```

Step 4: Fill the Tree

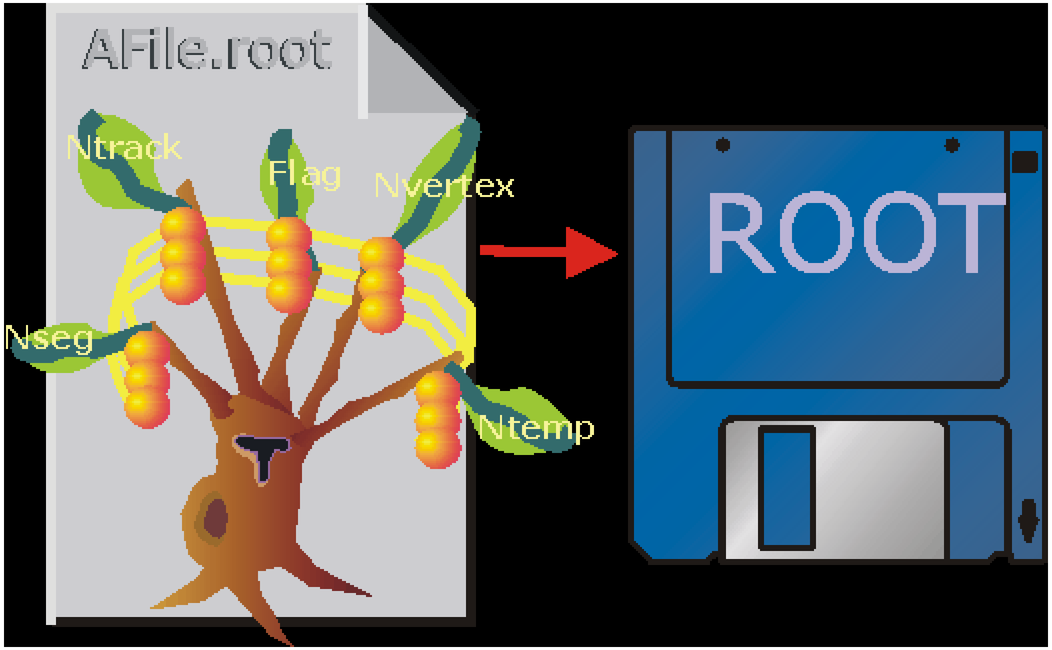
- Create a for loop
- Assign values to the object contained in each branch
- TTree::Fill() creates a new entry in the tree: snapshot of values of branches' objects




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

Step 5: Write Tree To File

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



Writing a Tree:a complete example

```
void tree1w() {
  // create a tree file tree1.root - create the file,
  // the Tree and a few branches
  TFile f("tree1.root","recreate");
  TTree t1("t1","a simple Tree with simple variables");
  Float_t px, py, pz;
  Double_t random;
  Int_t ev;
  t1.Branch("px",&px,"px/F");
  t1.Branch("py",&py,"py/F");
  t1.Branch("pz",&pz,"pz/F");
  t1.Branch("ev",&ev,"ev/I");
  // fill the tree
  for (Int_t i=0; i<10000; i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    t1.Fill();
  }
  // save the Tree heade; the file will be automatically closed
  // when going out of the function scope
  t1.Write();
}
```

Reading a Tree: a complete example

```
void tree1r() {
  TFile *f = new TFile("tree1.root");
  TTree *t1 = (TTree*)f->Get("t1");
  Float_t px, py, pz;
  Double_t random;
  Int_t ev;
  t1->SetBranchAddress("px",&px);
  t1->SetBranchAddress("py",&py);
  t1->SetBranchAddress("pz",&pz);
  t1->SetBranchAddress("random",&random);
  t1->SetBranchAddress("ev",&ev);
  // create two histograms
  TH1F *hpx = new TH1F("hpx","px distribution",100,-3,3);
  TH2F *hpxpy = new TH2F("hpxpy","py vs px",30,-3,3,30,-3,3);
  // read all entries and fill the histograms
  Int_t nentries = static_cast<Int_t>(t1->GetEntries());
  for (Int_t i=0; i < nentries; i++) {
    t1->GetEntry(i);
    hpx->Fill(px);
    hpxpy->Fill(px,py);
  }
  // We do not close the file. We want to keep the generated histograms
  // we open a browser and the TreeViewer
  if (gROOT->IsBatch()) return;
  new TBrowser ();
  t1->StartViewer();
}
```

Adding a new branch to an existing Tree

```
void tree3AddBranch() {
  TFile f("tree3.root","update");
  Float_t new_v;
  TTree *t3 = dynamic_cast<TTree*>(f->Get("t3"));
  TBranch *newBranch = t3->Branch("new_v",&new_v,"new_v/F");

 // read the number of entries in the t3
  Int_t nentries = static_cast<Int_t>(t3->GetEntries());
  for (Int_t i = 0; i < nentries; i++){
    new_v = gRandom->Gaus(0,1);
    newBranch->Fill();
  }
  t3->Write("",TObject::kOverwrite); //save only the new
                                     // version of the tree
}
```

Importing an ASCII file

```
{
  gROOT->Reset();
  TFile *f = new TFile("basic2.root","RECREATE");
  TTree *T = new TTree("ntuple","data from ascii file");
  Long64_t nlines = T->ReadFile("basic.dat","x:y:z");
  printf(" found %lld points\n",nlines);
  T->Draw("x","z>2");
  T->Write();
}
```

TTree Selection Syntax

- Prints the first 8 variables of the tree

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

- Prints all the variables of the tree. Specific variables of the tree can be explicit selected by list them in column separated list

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

- Prints the values of var1, var2 and var3. A selection can be applied in the second argument

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

- Prints the values of var1, var2 and var3 for the entries where var1 is exactly 0

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

Using TCut with TTree::Draw

- A TCut is a specialized string object used for TTree selections
  - inherits from TNamed, i.e has a name and a title
  - adds a set of operators to do logical string concatenation.
  - operators =, +=, +, *, !, &&, || are overloaded,

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

Chains of Trees

- A TChain is a collection of TTrees
  - a TChain is-a TTree
- Same semantics for TChains and TTrees

```
root [] .x chain.C
root [] chain.Draw(...);
root [] chain.Process(“analysis.C”);
root [] chain.MakeClass();
```

```
void chain(){
  TChain chain("MyTree");
  chain.Add("files/test_1.root");
  chain.Add("files/test_2.root");
  chain.Add("files/test_3.root");
  chain.Add("files/test_4.root");
}
```

Writing User Objects to a TTree

MyTrack.h

```
class MyTrack : public TLorentzVector {
public:

  MyTrack():  {}
  virtual ~MyTrack();

  float charge;
  float chiSq;
  int  nHits;

  ClassDef(MyTrack,1)
};
```

MyTrack.cc

```
#include "interface/MyTrack.h"
ClassImp(MyTrack)
MyTrack::~MyTrack(): TLorentzVector(0,0,0,0),
                    charge(0.0),chiSq(0.0),nHits(0){}
```

TClonesArray

- Array of objects of the same class ("clones")
- Designed for repetitive data analysis tasks: same type of objects created and deleted many times

TreeMaker & TClonesArray

```
Class TreeMaker : public TObject {
  TClonesArray* cloneEvent;
  TClonesArray* cloneMet;
  TClonesArray* cloneTrack;
  TClonesArray* cloneElectron;
  TClonesArray* cloneMuon;
  TClonesArray* cloneJet;

  EventInfo* eventB; // run, event etc.
  MyMet* metB;
  MyTrack* trackB;
  MyElectron* eleB;
  MyMuon* muonB;
  MyJet* jetB;
  int fnEle;
  int fnMuon;
  int fnJet;

  TFile *file;
  TTree *tree;

  ClassDef(TreeMakerModule,1)
};
```

TreeMaker & TClonesArray

```
TreeMaker::init(){
  file = TFile::Open(“test.root”, "RECREATE", "Skimmed Ntuple");
  tree = new TTree("RTree", "RTree");

  cloneEvent = new TClonesArray("EventInfo");
  tree->Branch("EventInfo", &cloneEvent, 32000, 2);

  cloneJet = new TClonesArray("MyJet");
  tree->Branch("MyJet", &cloneJet, 32000, 2);
  tree->Branch("nJet", &fnJet, "fnJet/I");
  ...
}
TreeMaker::Loop(){
  cloneEvent->Clear();cloneMet->Clear();
  cloneTrack->Clear(); cloneJet->Clear(); etc.

  eventB = new ( (*cloneEvent)[0] ) EventInfo();
  eventB->run = run;
  eventB->evt = event;
  for (int i=0; i <= njet; ++i) {
    jetB = new ((*cloneJet)[fnJet++]) MyJet();
    jetB->EMFraction = frac;
  }
  tree->Fill();
}
```

Store STL vectors in a TTree

```
class MyEvent : public TObject {
  ClassDef(MyEvent,1) // The macro

public:
  MyEvent();
  virtual ~MyEvent();

  int run;
  int event;
  // L1 Trigger decision and bits
  bool theL1accept;
  int theL1SingleTauBit;
  int theL1DoubleTauBit;
  // HL objects
  vector<MyJet>       genJets;
  vector<MyJet>       jets;
  vector<MyTrack>     tracks;
  vector<MyCluster>   clusters;
  vector<MyElectron>  electrons;
  vector<MyMuon>      muons;
  void clear() {
    genJets.clear();
    jets.clear();
    tracks.clear();
    clusters.clear();
    electrons.clear();
    muons.clear();
  }
};
```

LinkDef.h

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ class MyJet+;
#pragma link C++ class vector<MyJet>;
#pragma link C++ class MyCluster+;
#pragma link C++ class vector<MyCluster>;
#pragma link C++ class MyTrack+;
#pragma link C++ class vector<MyTrack>;
#pragma link C++ class MyEvent+;
#pragma link C++ class vector<MyTauProduct>;
#pragma link C++ class MyTauProduct+;
#endif
```

libMyEvent.so

Store STL vectors in a TTree

```
Class MyModule: public edm::EDAnalyzer    // interface/MyModule.h
public:
  explicit MyModule(const edm::ParameterSet&);
  ~MyModule() {}
  virtual void analyze(const edm::Event& iEvent,
                       const edm::EventSetup& iSetup);
private:
  .......
  TTree* _tree;
  TFile* _file;
  MyEvent* _event;
};
                                          // src/MyModule.cc
void MyModule::beginJob(const edm::EventSetup& iSetup) {
  _file = TFile::Open("test.root", "RECREATE");
  _tree = new TTree ("TT","Tree with Events");
  _tree->SetAutoSave(1000000000);

  _event = new MyEvent();
  int bufsize = 256000;
  int split   = 1;
  _tree->Branch("EventBranch","MyEvent",&_event,bufsize,split);
}
```

Store STL vectors in a TTree

```
MyModule::analyze(const Event& iEvent, const EventSetup& iSetup)
{
  // read event and prepare information
  // Create the event object
  MyEvent* ev = new MyEvent();

  // Event quantities
  ev->run    = irun;
  ev->event = ievent;
  ev->L1accept       = L1accept;
  ev->L1SingleTauBit = SingleTauBit;
  ev->L1DoubleTauBit = DoubleTauBit;

  // collections
  ev->genJets    = genJets;
  ev->jets       = jets;       // vector<MyJets>
  ev->tracks     = tracks;
  ev->clusters   = clusters;
  ev->electrons = electrons;
  ev->muons      = muons;
  _event = ev;
  _tree->Fill();
}
```

Save the TTree

```
void MyModule::endJob(const edm::EventSetup& iSetup) {
  _file->cd();
  _tree->Print(); // show headers etc.
  _tree->Write();
}
```

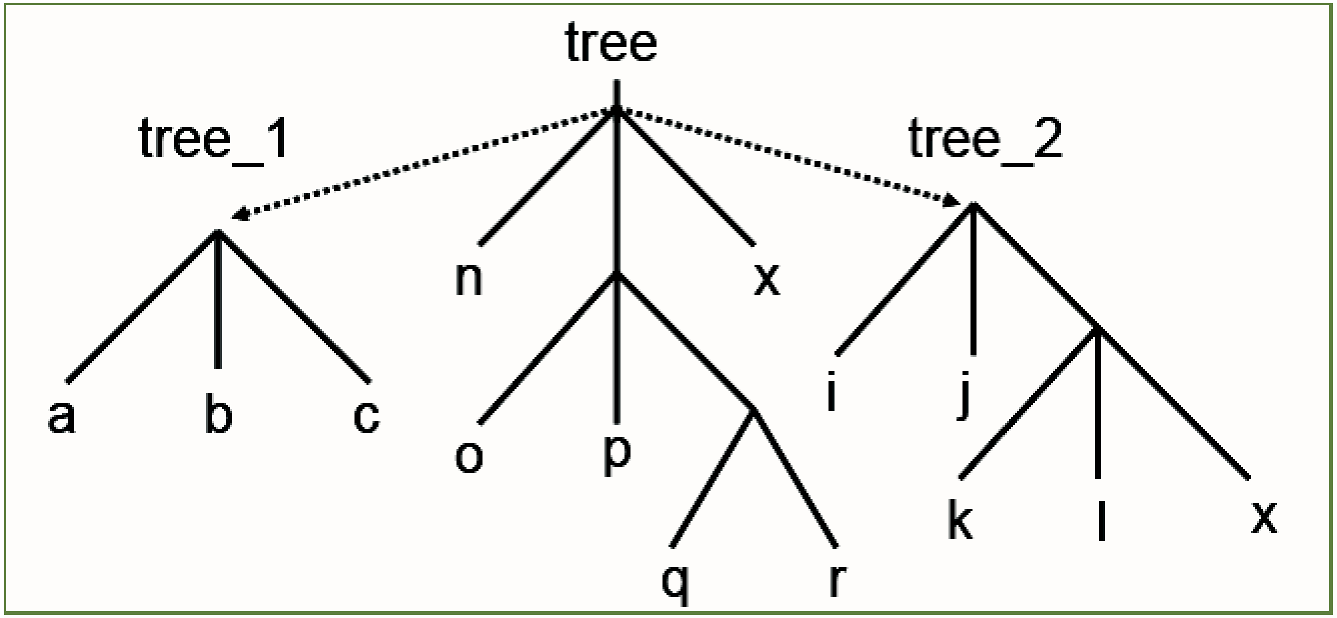
TTree Analysis

```
root [0] .L libMyEvent.so // without which the following will happen
root [1] TFile* file = Tfile::Open(“test.root”);
Attaching file test.root as _file0...
Warning in <TClass::TClass>: no dictionary for class MyEvent is available
Warning in <TClass::TClass>: no dictionary for class MyJet is available
Warning in <TClass::TClass>: no dictionary for class TLorentzVector is available
Warning in <TClass::TClass>: no dictionary for class TVector3 is available
Warning in <TClass::TClass>: no dictionary for class MyTrack is available
Warning in <TClass::TClass>: no dictionary for class MyCluster is available
Warning in <TClass::TClass>: no dictionary for class MyTauProduct is available

root [2] _file0->ls()
TFile**          test.root
 TFile*          test.root
  KEY: TTree     MyTree;1      Tree with Events

root [3] TTree *t = dynamic_cast<TTree *>(_file0->Get("MyTree"))
root [4] t->MakeClass()
Info in <TTreePlayer::MakeClass>: Files: MyTree.h and MyTree.C generated from
TTree: MyTree

root [] .L MyTree.C++
root [] MyTree t
root [] t.GetEntry(12); // Fill t data members with entry number 12
root [] t.Show();       // Show values of entry 12
root [] t.Show(16);     // Read and show values of entry 16
root [] t.Loop();       // Loop on all entries
```

Tree Friends

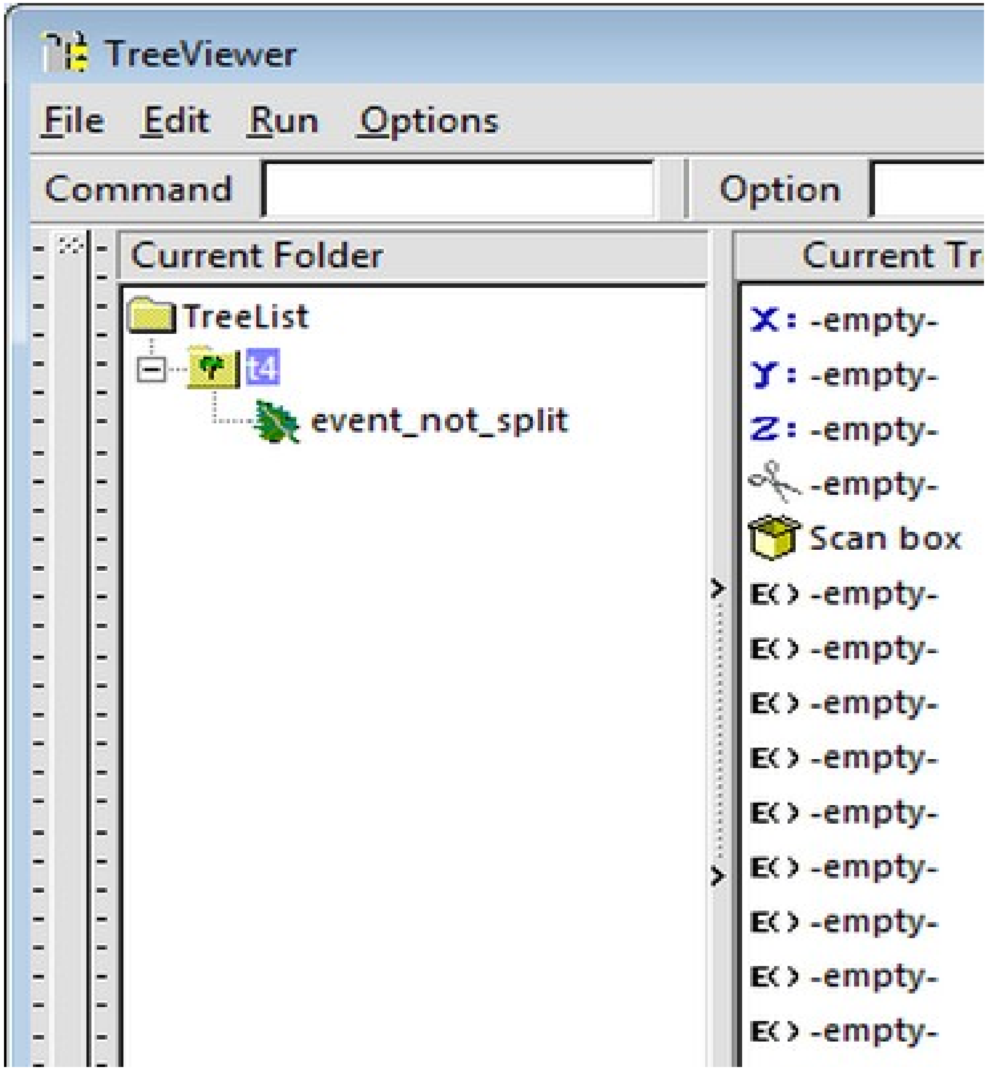
- Trees are designed to be read only
- Often, people want to add branches to existing trees and write their data into it
- Using tree friends is the solution
  - create a new file holding the new tree
  - create a new Tree holding the branches for the user data
  - fill the tree/branches with user data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

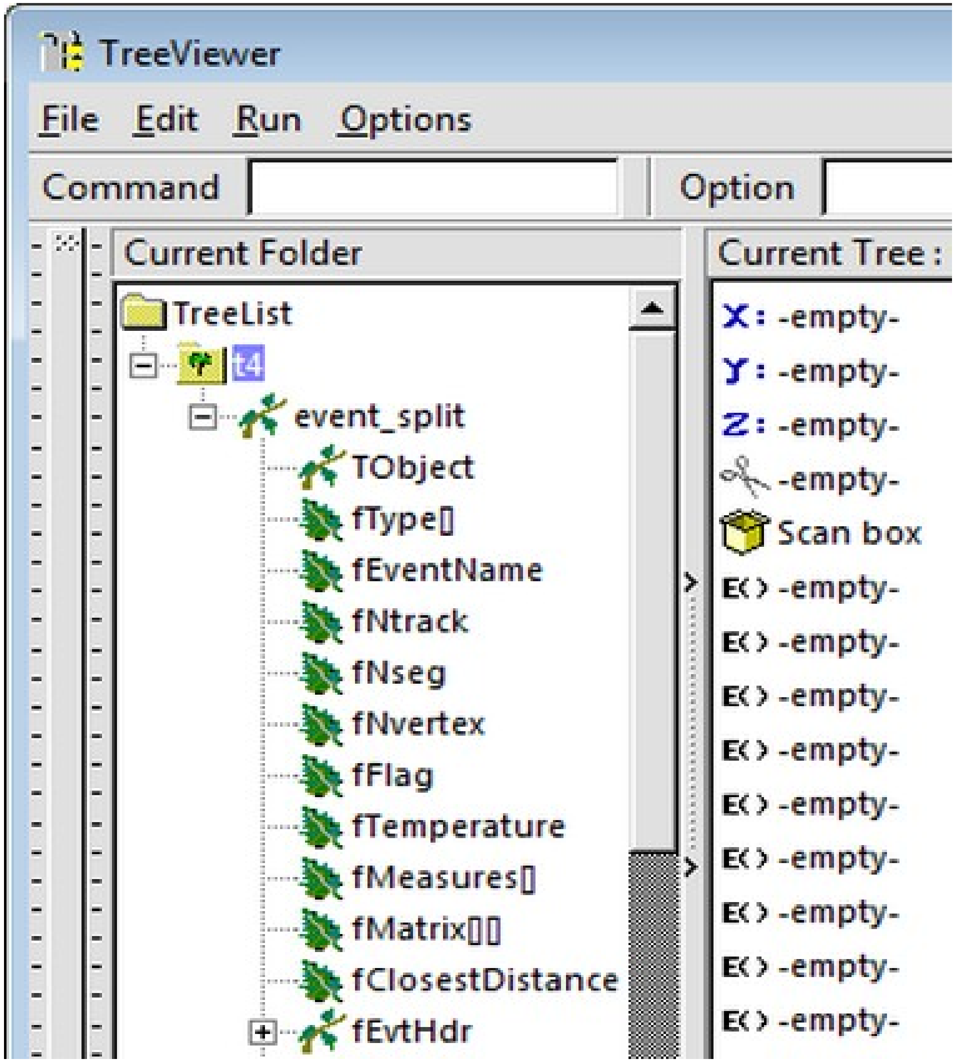
Tree Friends




```
TFile f1("tree.root");
// get hold of the TTree
tree.AddFriend("tree_1", "tree1.root");
tree.AddFriend("tree_2", "tree2.root");
tree.Draw("x:a", "k<c");
tree.Draw("x:tree_2.x");
```

Splitting



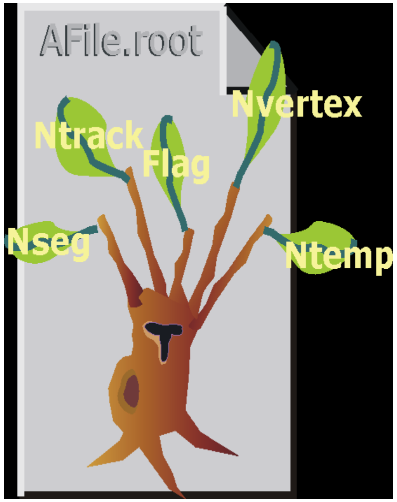


Split level = 0

Split level = 99

Splitting

- 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
- Supports STL containers too, even vector<T*>!



Split level = 0

split level = 99 (default)

```
tree->Branch("EvBr", &event, 64000,split_level);
```

Splitting: 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

Summary: Trees

- TTree 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