ROOT: Basics

PAVAN POOT PANDEY

DEPARTMENT OF PHYSICS & ASTROPHYSICS

(UNIVERSITY OF DELHI)

ROOT

ROOT is an OO(Object Oriented) C++ framework developed for large scale data handling that provides :

•an efficient data storage and access system designed to support structured datasets of Peta Byte scale
• a C++ interpreter
•Histogramming and fitting
•advanced statistical analysis algorithms (multi dimensional histograms, fitting, minimization, cluster finding etc.)

- Thanks to the embedded CINT C++ interpreter, both command line and scripting language is C++
- The ROOT library can be accessed seamlessly from Python/Ruby as well.

Object-Oriented Programming offers considerable benefits compared to Procedure-Oriented Programming.

The user Interacts with ROOT via

A graphical user interface

The command line

C ++ scripts

Compiled programs etc.

Installing ROOT

To install ROOT one will need to go to the ROOT website at: http://root.cern.ch/root/Availability.html. Then one have
a choice to download the binaries or the source. The source is quicker to transfer since it is only ~22 MB.
The binaries compiled with no degug information range from ~35 MB to ~45MB.

Before downloading a binary version make sure your machine contains the right run-time environment. In most cases it is not possible to run a version compiled with, e.g., gcc4.0 on a platform where only gcc 3.2 is installed.
In such cases one'll have to install ROOT from source.

How to Find More Information

- The ROOT web site has up to date documentation. The ROOT source code automatically generates this documentation, so each class is explicitly documented on its own web page, which is always up to date with the latest official release of ROOT.
- The ROOT Reference Guide web pages can be found at http://root.cern.ch/root/html/ClassIndex.html.
- Each page contains a class description, and an explanation of each method. It shows the class inheritance tree and lets you jump to the parent class page by clicking on the class name.

ROOT Library Structure

- ➔ROOT libraries are arranged in a layered structure.
- ➔The Core classes are always required (support for Run Time Type Information, basic I/O & interpreter).
- ➔The optional libraries (you load only what you use)
- ➔Why shared libraries?
- ➔reduce the application link time
- ➔reduce the application size
- ➔can be used with other class libraries
- ➔usually loaded via the plug-in manager

TObject – The Base Class

- TObject provides default behavior and protocol for almost all objects in the ROOT system.
- object I/O (Read(), Write())
- error Handling (Warning(), Error(), Fatal())
- sorting (Compare(), IsEqual())
- inspection(Dump(), Inspect())
- drawing, printing
- bit handling (SetBit(), TestBit())
- An object of any class that inherits from TObject can be made persistent (object I/O)
-

TROOT

- the TROOT object is the main entry point to the system
- –created as soon as the Core library gets loaded
- –initializes the rest of the ROOT system
- –a singleton, accessible via the global pointer gROOT
- An omnipotent global, handle with care
- provides many global services
- gROOT->GetListOfFiles()
- gROOT->GetListOfCanvases()

- via gROOT you can find basically every object

  created by the system,

- TH1F *hpx = (TH1F*) gROOT->FindObject(“hpx”) // C-style
- TH1F *hpx = dynamic_cast<TH1F*>(gROOT->FindObject(“hpx”)) // C++ style

Latex Support

```
#include "TROOT.h"
#include "TCanvas.h"
#include "TLatex.h"
void latex()
{
gROOT->Reset();
TCanvas *c1 = new TCanvas("c1");
TLatex l;
l.SetTextAlign(23);
l.SetTextSize(0.1);
l.DrawLatex(0.5,0.95,"e^{+}e^{-}#rightarrowZ^{0}#
rightarrowI#bar{I},
q#bar{q}");
c1->Print("latex2.ps");
}
```

Physics analysis using ROOT

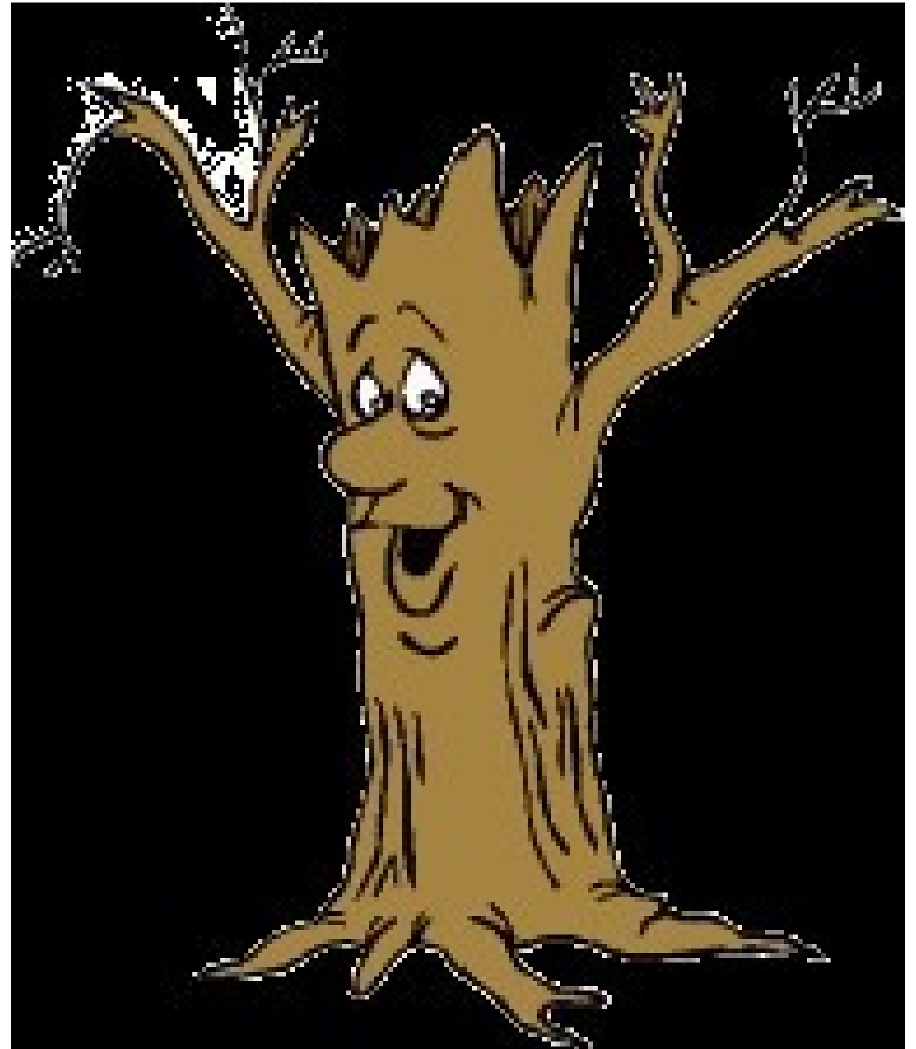
- HEP analysis performed mainly with ROOT
- there are several ways:-
- simple macro to compiled classes
- TNtuple, Ttree, Tchain
- TClonesArray, TSelector, Tcut
-
-

  (TNtuple is simple tree restricted to a list of float variables only.Each variable goes to a separate branch. We'll discuss Ttree in detail.)

▪

Trees

- Efficient storage and access for huge amounts of structured data
- a) allows selective access of data
- b) TTree knows its layout



•Trees allow direct and random access to any entry

—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.

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");
TTree *t = new TTree("myTree","A Tree");
Event *myEvent = new Event();
t->Branch("EventBranch", &myEvent);
for (int e=0;e<100000;++e) {
myEvent->Generate(); // hypothetical
t->Fill();
}
t->Write();
}
```

-

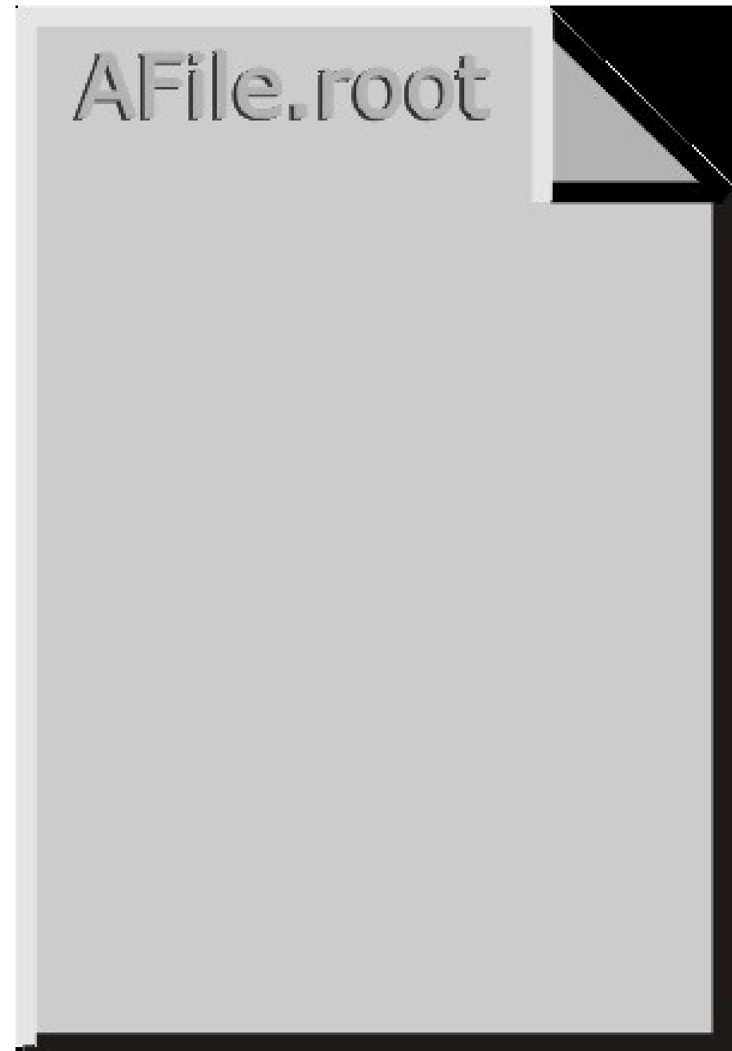
Step 1: Create a TFile Object

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

e.g.

TFile *hfile = TFile::Open("AFile.root",

"RECREATE");



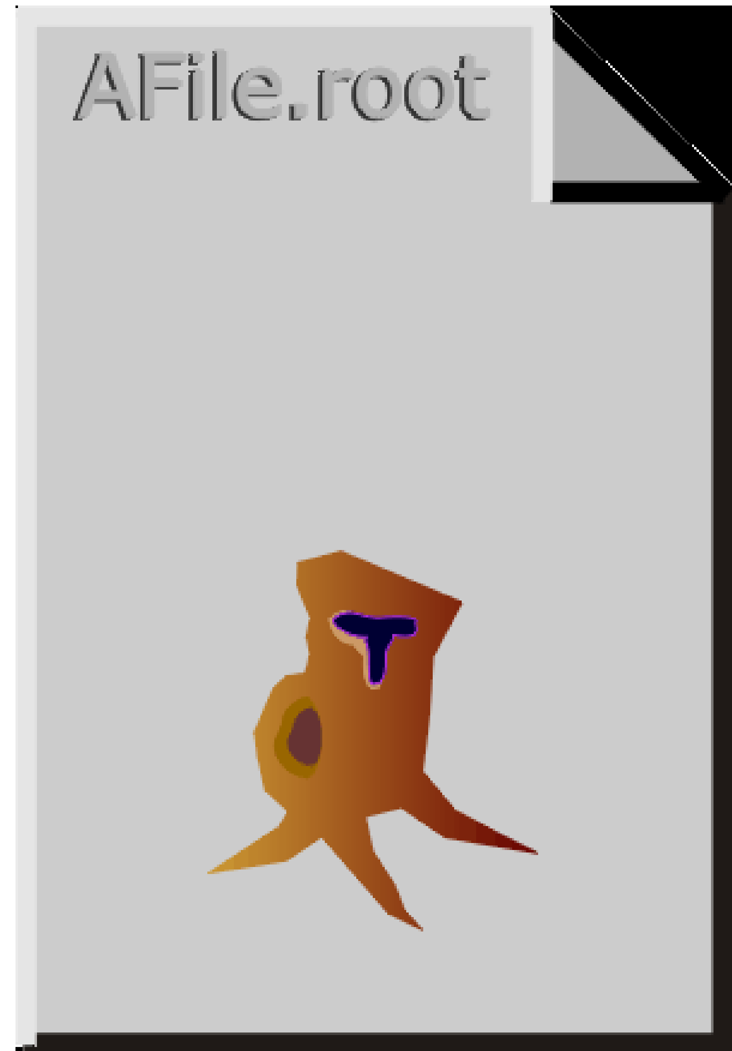
Step 2: Create a TTree Object

- The TTree constructor

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

  - Tree title

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



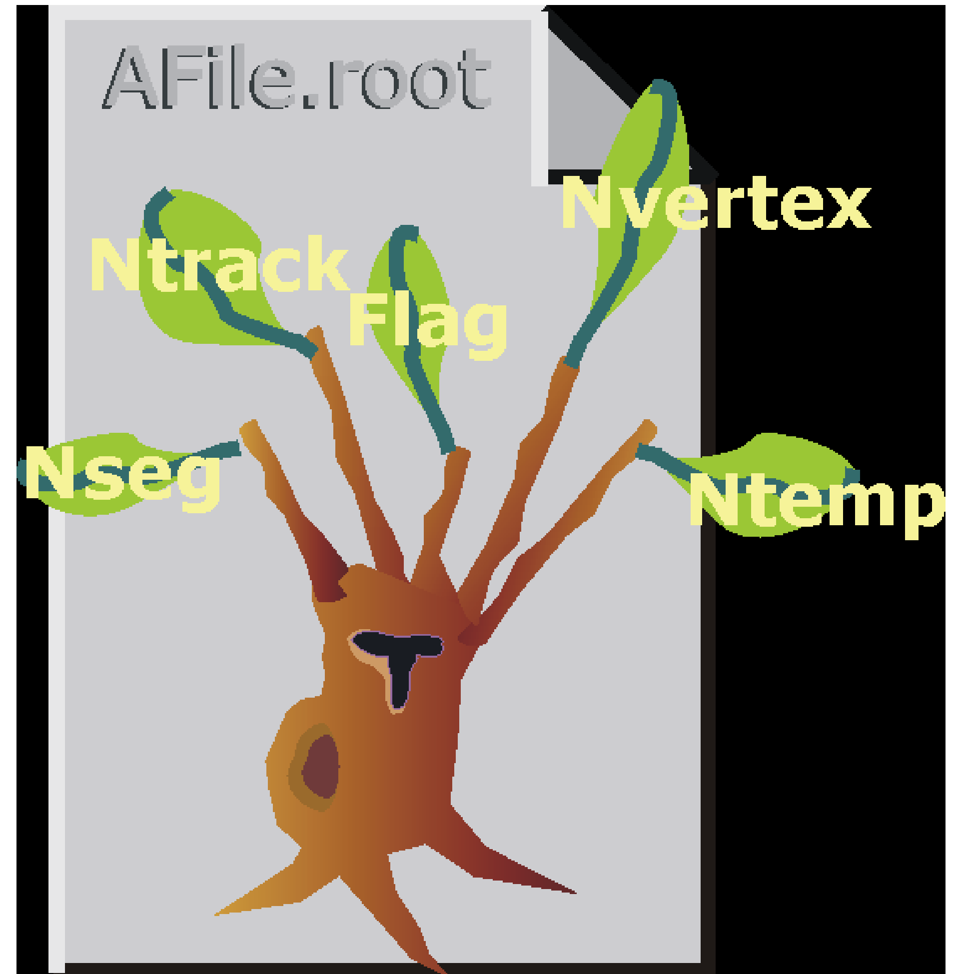
Step 3: Adding a Branch

- Branch name
- Address of pointer to the object

  Event *myEvent = new Event();

  myTree->Branch("eBranch",&myEvent);

-

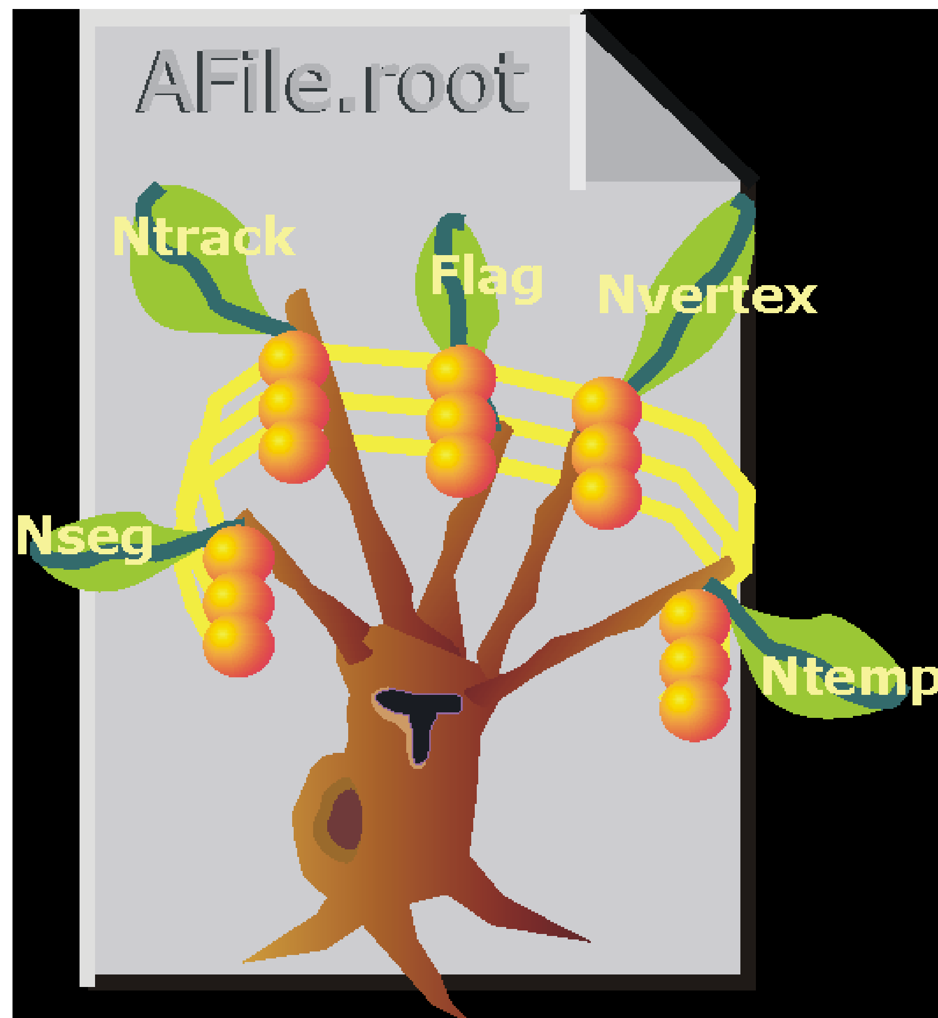


Step 4: Fill the Tree

- 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

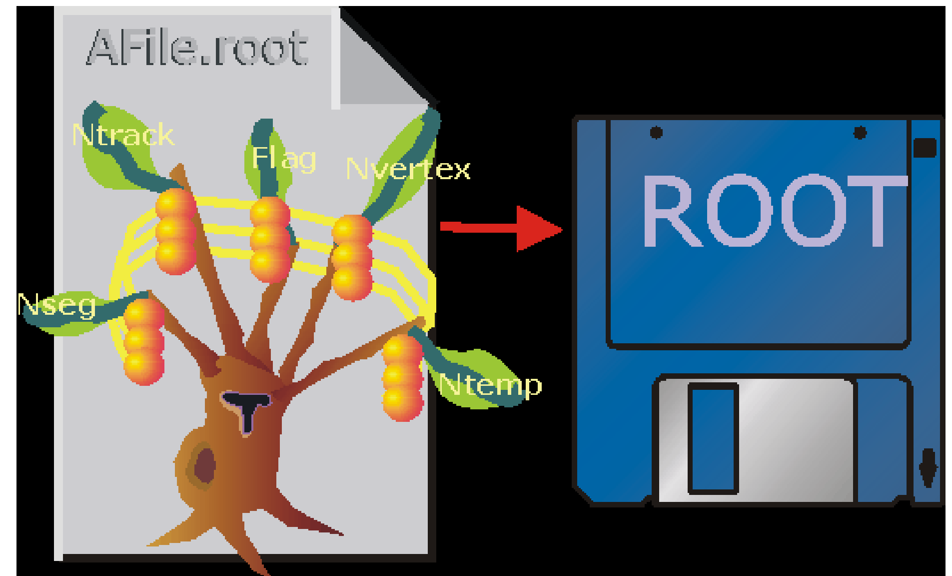
```
for (int e=0;e<100000;++e) {
myEvent->Generate(e); //fill event
myTree->Fill(); // fill the tree
}
```

-



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();
}
```

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
  – using 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

THANK YOU