Introduction to ROOT

Asif Saddique
National Centre for Physics (NCP), Islamabad, Pakistan

ROOT Tree, A useful Tool for Data Analysis
Tutorial # 3

August 25, 2016

5th School on LHC Physics
Why Tree?

Defining a tree in useful because:

- you can store complex types of data, i.e. objects can be stored in a tree.
- ROOT tree is extremely efficient write once, read many times.
- All the variable stay connected for all the entries. You can easily change selection criteria in a small macro.
- Trees allow fast direct and random access to any entry.
  - Trees have column-wise access. They can directly access to any event, any branch or any leaf even in the case of variable length structures.
  - 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!
- Trees are Optimized for network access, and they are buffered to disk.
A tree (TTree) contains branches (TBranch) and leaves (TLeaf).

Figure: Examples of split, non-split trees and tree with a branch containing several leaves (leaves).
Wait !!! Before Tree, Get Concept of Cuts/Filters

In data analysis many variables are linked/dependent to/on each other.

<table>
<thead>
<tr>
<th>Cuts/Filters</th>
<th>#Tracks</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Cut</td>
<td>5</td>
</tr>
<tr>
<td>$\theta &lt; 60^\circ$</td>
<td>3</td>
</tr>
<tr>
<td>$P_x &lt; 10$</td>
<td>3</td>
</tr>
<tr>
<td>$P_y &gt; 10$</td>
<td>4</td>
</tr>
<tr>
<td>$\theta &gt; 45^\circ$</td>
<td>3</td>
</tr>
<tr>
<td>$45^\circ &lt; \theta &lt; 60^\circ$</td>
<td>1</td>
</tr>
<tr>
<td>$\theta &lt; 60^\circ$ &amp; $P_x &gt; 10$</td>
<td>2</td>
</tr>
<tr>
<td>$P_x &lt; 9$ &amp; $P_y &gt; 16$</td>
<td>2</td>
</tr>
<tr>
<td>$P_x \leq 9$ &amp; $P_y &gt; 16$</td>
<td>2</td>
</tr>
<tr>
<td>$P_x \leq 9$ &amp; $P_y \geq 16$</td>
<td>3</td>
</tr>
<tr>
<td>$P_x \leq 9$ &amp; $P_y \geq 16$ &amp; $\theta &gt; 60^\circ$</td>
<td>2</td>
</tr>
</tbody>
</table>

Your supervisor gives you a task in the morning, and you store $P_x$ in one hitso, and $P_y$ in second hitso (after 5hr code running), for $\theta < 60^\circ$. In evening he says, no no its wrong, you need $P_x$ and $P_y$ histograms, for $\theta < 45^\circ$. WHAT!!!! I need another 5 hr to run the code again ........

BUT DON’T WORRY STAY TUNED
Writing a Tree

- A tree is defined as:
  ```c
  TTree *mytree = new TTree("ntuples","an example Tree");
  ```

- A branches in this tree can be defined as:
  ```c
  mytree->Branch("px",&px,"px/F");
  ```

  Here, the branch variable “px” (a leaf) must be defined before setting up branch.

- Fill the tree in event loop.
  ```c
  for (Int_t evt=0; evt<1000; evt++) {
    px = gRandom->Gaus(0,2);
    mytree->Fill();
  }
  ```

- After the event loop, any leaf histogram can drawn with any cut.
  ```c
  mytree->Draw("px","px>2");
  ```

  But here binning is automatic, we look into this matter later.
Writing a Tree

Let’s write several branches in a tree and put it into a root file.

```c
void ATree_with_ThreeBranches()
{
   // ***The code starts****
   const Int_t kMaxTrack = 500;
   // Defining branch variables
   Int_t ntrack; Float_t px, py;
   // Creating a root file to put the tree in
   TFile file("mybranches.root","recreate");
   // Creating a tree
   TTree *mytree = new TTree("ntuples","an example Tree");
   // Creating branches in the tree
   mytree→Branch("ntrack",&ntrack,"ntrack/I");
   mytree→Branch("px",&px,"px/F");
   mytree→Branch("py",&py,"py/F");
   for (Int_t evt=0; evt<1000; evt++)
   {
      // ***Event loop starts***
      Int_t nt = gRandom→Rndm()*(kMaxTrack-1);
      px = gRandom→Gaus(0,2);
      py = gRandom→Gaus(1,2);
      ntrack = nt;
      mytree→Fill();
   } // ***Event loop ends***
   mytree→Draw("ntrack","px<4");
} // ***The code ends****
```

The above code defines a tree with three branches, and writes them into “mybranches.root”, and draws the leaf histogram for “track” for “px<4” (so we started getting rewards !!!)
void ATree_with_ThreeBranches()
{
  // ***The code starts****
  const Int_t kMaxTrack = 500;
  // Defining branch variables
  Int_t ntrack; Float_t px, py;
  // Defining/opening an ascii file
  ofstream outFile;
  outFile.open("myAscii.dat");
  // Creating a root file to put the tree in
  TFile file("mybranches.root","recreate");
  // Creating a tree
  TTree *mytree = new TTree("ntuples","an example Tree");
  // Creating branches in the tree
  mytree->Branch("ntrack","ntrack","ntrack/I");
  mytree->Branch("px","px","px/F");
  mytree->Branch("py","py","py/F");
  for (Int_t evt=0; evt<1000; evt++)
  {
    // ***Event loop starts***
    Int_t nt = gRandom->Rndm()*(kMaxTrack-1);
    px = gRandom->Gaus(0,2);
    py = gRandom->Gaus(1,2);
    ntrack = nt;
    mytree->Fill();
    outFile << ntrack << " " << px << " " << py << endl;
  } // ***Event loop ends***
  outFile.close();
  mytree->Draw("ntrack","px<4");
} // ***The code ends****

- Blue lines are added in the previous code to make an ascii file.
- The code generates “myAscii.dat” file, which contains three columns.

What a tree has to do with an ascii file?? see later!!!

Let’s worry about “mybranches.root” for now.
Browsing a Tree

We can check the created tree by TBrowser.

- First connect the root file to prompt:
  
  ```
  $ root mybranches.root
  ```

  Alternatively, you can also load the root file in prompt.

- Then open TBrowser:

  ```
  root[ ] new TBrowser
  ```
void ReadTreeMakeHisto() {
   // Reading the root file
   TFile *file = new TFile("/PathToRootFile/mybranches.root", "READ");
   // Go into the file
   file->cd();
   // Calling branches, and define bins you want
   // Putting cuts/set-of-cuts on branches
   mytree->Draw("ntrack>>Track px upto 4(100,10,500)\" , \"px<4\");
   mytree->Draw("ntrack>>Track py upto 3(100,10,500)\" , \"py<3\");
   mytree->Draw("ntrack>>Track px4 py3(100,10,500)\" , \"px<4 && py<3\");
   // Defining Histograms and connecting them with tree branches
   TH1F *Track_4x = (TH1F*)gDirectory->Get("Track px upto 4");
   TH1F *Track_3y = (TH1F*)gDirectory->Get("Track py upto 3");
   TH1F *Track_xy = (TH1F*)gDirectory->Get("Track px4 py3");
   // Drawing an example Histogram
   Track_4x->Draw();
   // Creating a root file to put histos obtained from tree
   TFile hfile("myHistofromTree.root","recreate");
   // Making a directory inside root file
   hfile.mkdir("Histo");
   // Going inside directory
   hfile.cd("Histo");
   // Writing histos inside directory
   Track_4x->Write();
   Track_3y->Write();
   Track_xy->Write();
}

- The code generates “myHistofromTree.root” file, and also draws an example plot:

Please note statistics from the stat box.
A Tree can easily read an Ascii file

The code generates “basic.root” file with a tree “ntuples” containing three branches, “tracks”, “px” and “py”. It also produces following plot:

The plot obtained from Ascii file through tree is obtained by using the same binning and selection criteria as used for the plot on previous slide. The stat box shows the same results. Hence a tree can efficiently read an ascii file.
Printing a Tree

- First load the root file in prompt:
  ```cpp
  root[ ] TFile *file=new TFile("mybranches.root");
  ```

- Check if tree is there in the file:
  ```cpp
  root[ ] file → ls()
  ```

- To print information from a tree:
  ```cpp
  root[ ] mytree → Print()
  ```

It will print the tree structure (sizes, branches, entries etc.) as following:

```
*Tree :mytree : an example Tree
*Entries : 1000 : Total = 13988 bytes File Size = 10207
*          : Tree compression factor = 1.26

*0 :ntrack : ntrack/I
*Entries : 1000 : Total Size= 4553 bytes File Size = 2012
*          : Basket Size= 32000 bytes Compression= 2.03
*          : px/F
*Entries : 1000 : Total Size= 4533 bytes File Size = 3830
*          : Basket Size= 32000 bytes Compression= 1.06
*          : py/F
*Entries : 1000 : Total Size= 4533 bytes File Size = 3816
*          : Basket Size= 32000 bytes Compression= 1.07
```
To scan information from a tree:

```
root[] mytree → Scan()
```

It will print the structure of each entry as following:

```
+---------------------+---------------------+---------------------+---------------------+---------------------+
| Row | ntrack | px | py |
+---------------------+---------------------+---------------------+---------------------+
| 0 | 490 | -0.869528 | 2.5635924 |
| 1 | 115 | -5.252603 | -0.801751 |
| 2 | 369 | 0.0158244 | 0.1784736 |
| 3 | 157 | 0.3833004 | -0.970132 |
| 4 | 84 | -3.569256 | -0.147238 |
| 5 | 15 | -2.776595 | 2.5347931 |
| 6 | 289 | 1.1594425 | 0.2357311 |
| 7 | 248 | -1.631610 | 0.1861021 |
| 8 | 58 | 2.3306736 | 0.0915950 |
| 9 | 361 | -0.999278 | 0.6351276 |
| 10 | 49 | -0.485695 | 4.9937224 |
| 11 | 64 | 4.5208911 | 4.0810632 |
| 12 | 63 | -1.741184 | 4.0493841 |
| 13 | 145 | 1.1042200 | 0.7273818 |
| 14 | 70 | 3.2522122 | 0.7632834 |
| 15 | 22 | 0.0247242 | 1.0441280 |
| 16 | 369 | -2.643235 | 4.4886541 |
| 17 | 338 | -0.494545 | -1.360430 |
| 18 | 169 | 3.5270268 | -3.019523 |
| 19 | 184 | 0.6015881 | 1.9763412 |
| 20 | 414 | 0.4878255 | -1.184678 |
| 21 | 305 | -1.575853 | -4.160276 |
| 22 | 21 | -2.486564 | 0.5242354 |
| 23 | 219 | -1.307872 | 0.0622450 |
| 24 | 215 | -1.670127 | 0.6595612 |
+---------------------+---------------------+---------------------+---------------------+
```

Type <CR> to continue or q to quit ==>
Making a Class from a Tree

- First load the root file in prompt:
  
  ```c
  root[ ] TFile *file=new TFile("mybranches.root");
  ```

- Cross check the tree name:
  
  ```c
  root[ ] file → ls()
  ```

- Now make your Class:
  
  ```c
  root[ ] mytree → MakeClass("MyCode");
  ```

It will show the output like following:

```c
Info in <TTreePlayer::MakeClass>: Files: MyCode.h and MyCode.C generated from TTree: mytree
```

Here **MyCode.C** contains the basic structure of code with an event loop, and **MyCode.h** tells you variable that you can access while building your code in the event loop (inside MyCode.C).

**Remember, Its good way to start your code.**
Each node is the branch in Tree
Tree Memory

- Each node is the branch in Tree
Each node is the branch in Tree
Tree Memory

- Each node is the branch in Tree
Tree Memory

- Each node is the branch in Tree
If there are three root files, “file1.root”, “file2.root” and “file3.root”, which have the same tree ”T”. It possible to combine them by TChain:

```c
TChain chain("T")
chain.Add("file1.root");
chain.Add("file2.root");
chain.Add("file3.root");
```

TChain can be used like TTree.
If there are three root files, “file1.root”, “file2.root” and “file3.root”, which have the same tree ”T”. It is possible to combine them by TChain:

```cpp
TChain chain("T")
chain.Add("file1.root");
chain.Add("file2.root");
chain.Add("file3.root");
```

TChain can be used like TTree.
Making a Class from a Tree

For example, you have data files in root format and you want to analyze the data in those files. Take any data file and make a class from the tree to start your code.

Preview of “MyCode.C”

```c
#include MyCode_cxx
#include "MyCode.h"
#include <TCanvas.h>
#include <TStyle.h>

void MyCode::Loop()
{
    // In a ROOT session, you can do:
    // Root > .L MyCode.C
    // Root > MyCode 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.GetEntries(); // Loop on all entries
    //
    // This is the loop skeleton where:
    // jentry is the global entry number in the chain
    // jentry is the entry number in the current Tree
    // Note that the argument to GetEntry must be:
    // jentry for TChain::GetEntry
    // jentry for TTree::GetEntry and TBranch::GetEntry
    //
    // To read only selected branches, Insert statements like:
    // METHOD1:
    // fChain->SetBranchStatus("\"\",0); // disable all branches
    // fChain->SetBranchStatus("\"branchName\",1); // activate branchname
    // METHOD2: replace line
    // fChain->GetEntry(jentry); // read all branches
    // by:
    // fChain->SetBranchStatus("\"branchName\",1); // read only this branch
    // if (fChain == 0) return;
    Long64_t nentries = fChain->GetEntriesFast();
    Long64_t nbytes = 0, nb = 0;
    for (Long64_t jentry=0; jentry<nentries;jentry++) {
        // Optional a Global Entry check
        if (jentry < 0) break;
        nb = fChain->GetEntry(jentry); nbytes += nb;
        // if (Cut (jentry) < 0) continue;
    }
}
```

Preview of “MyCode.h”

```c
class MyCode {
public:
    TTree *fChain; // pointer to the analyzed TTree or TChain
    Int_t fCurrent; // current Tree number in a TChain
    // Declaration of leaf types
    Int_t ntrack;
    Float_t px;
    Float_t py;
    // List of branches
    TBranch *b_ntrack; //
    TBranch *b_px; //
    TBranch *b_py; //

    MyCode(TTree *ttree=0);
    virtual ~MyCode();
    virtual Int_t Cut(Long64_t entry);
    virtual Int_t GetEntry(Long64_t entry);
    virtual Long64_t LoadTree(Long64_t entry);
    virtual void Init(TTree *t);
    virtual void Loop();
    virtual void Notify();
}
```

Let’s focus on **Method1** to read the tree and write some code in event loop.
Making a Class from a Tree → Building/Running Code

For example, you have data files in root format and you want to analyze the data in those files. Take any data file and make a class from the tree to start your code.

Preview of “MyCode.C”

```c
#define MyCode_cxx
#include "MyCode.h"
#include <T2.h>
#include <TStyle.h>
#include <TCanvas.h>

void MyCode::Loop()
{
  // METHOD1: To Read Tree
  fChain->SetBranchStatus("=",0); // disable all branches
  fChain->SetBranchStatus("ntrack",1); // activate branch
  fChain->SetBranchStatus("px",1); // activate branch

  // Define HISTOs
  TH1 * px_100trk = new TH1("px_100trk",",", 50, 0,5);
  TH1 * px_200trk = new TH1("px_200trk",",", 50, 0,5);

  if (fChain == 0) return;
  Long64_t nentries = fChain->GetEntriesFast();

  Long64_t nbytes = 0, nb = 0;
  for (Long64_t jentry=0; jentry<nentries;jentry++)
  {
    Long64_t lentry = LoadTree(jentry);
    if (lentry < 0) break;
    nb = fChain->GetEntry(jentry); nbytes += nb;
    if (ntrack < 2) continue; // Throw this Event

    // FILLING HISTOS
    px_100trk->Fill(px);
    if (ntrack>100) px_200trk->Fill(px);
  }

  TFile* file = new TFile("output_MyCode.root", "RECREATE");
  file->cd();
  // WRITING HISTOS
  px_100trk->Write();
  px_200trk->Write();
}
```

The code generates **output_myCode.root** file with two histograms.

How to run the code “MyCode.C”

- Compile the code:
  ```
  root[] .L MyCode.C++;
  ```

  If there is no error, it will make **MyCode_C.so**

- Chain up all the input root files:
  ```
  root[] TChain* chain=new TChain("mytree");
  root[] chain->Add("mybranches.root");
  ```

- Load the shared object (so):
  ```
  root[] gSystem->Load("MyCode_C.so");
  ```

- Run the Loop:
  ```
  root[] MyCode run(chain);
  root[] run.Loop();
  ```
Exercises

● Writing/Reading a Tree

Exercise#1  Make/write a tree into a ROOT File for 800 entries containing $x$, $y$, $z$ and $t$ branches as floats. Please use $\{x, y, z, t\} = \{\text{Gaus}(0,1), \text{Gaus}(1,2), \text{Gaus}(1,3), \text{Gaus}(3,2)\}$. Draw variable $z$ for $x > 0$ and $t < 4$.

Exercise#2  Draw the variable $z$ again with the same conditions but with the bin range from 0 to 3 having 30 bins.

Exercise#3  Make a four columns ascii file containing variables $x$, $y$, $z$ and $t$. Also draw the same $z$ plot (as in Exercise 2) from the ascii file through a ROOT tree and compare the entries, mean and RMS values.

● TChain/combing trees from different File

Exercise#4  Run the above code 3 times but each time change the name of output ROOT file, e.g. myfile1.root, myfile2.root and myfile3.root. Join all the files by TChain method and find total number of entries [Hint: by using chain→GetEntries()] after combining all the three files.

● MakeClass/Building and Running the code

Exercise#5  Make a class from the ROOT file obtained from Exercise#1. Obtain a histogram for variable “$y$” for the case of “$t<3$” in a ROOT file.
Thanks