# Trees

March 17, 2017

In case you want to store large quantities of same-class objects, ROOT has designed the TTree and TNtuple classes specifically for that purpose. 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 on the other hand 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 with leaf data and the buffers are written to disk when it is full. Each object is not written individually, but rather collected and written a bunch at a time.

Since the unit to be compressed is a buffer, and the TTree contains many same-class objects, the header of the objects can be compressed. The TTree reduces the header of each object, but it still contains the class name. Using compression, the class name of each same-class object has a good chance of being compressed, since the compression algorithm recognizes the bit pattern representing the class name. Using a TTree and compression the header is reduced to about 4 bytes compared to the original 60 bytes. However, if compression is turned off, you will not see these large savings.

A tree uses a hierarchy of branches, and each branch can be read independently from any other branch.

## 0.1 A Simple TTree

Creates one branch with the TTree::Branch method. The first parameter of the Branch method is the branch name. The second parameter is the address from which the first leaf is to be read. In this example it is the address of the structure staff. Once the branch is defined, the script reads the data from the ASCII file into the staff_t structure and fills the tree. The ASCII file is closed, and the ROOT file is written to disk saving the tree.

## 0.2 Show an Entry with TTree::Show

Use the TTree::Show method to access one entry of a tree. For example to look at the 10th entry in the staff.root tree:

root[] TFile f("staff.root")

root[] T→Show(10)

## 0.3 Print the Tree Structure with TTree::Print

TTree::Print can be used to see the tree structure meaning the number of entries, the branches and the leaves.

## 0.4 Scan a Variable the Tree with TTree::Scan

TTree::Scan method shows all values of the list of leaves separated by a colon.

root[] T→Scan("Cost:Age:Children")

## 0.5 The Tree Viewer

The tree viewer is a quick and easy way to examine a tree. To start the tree viewer, open a file and object browser. Right click on a TTree and select StartViewer. You can also start the tree viewer from the command line. First load the viewer library.

root[] TFile f("staff.root")

root[] T→StartViewer()

If you want to start a tree viewer without a tree, you need to load the tree player library first:

root[] gSystem→Load("libTreeViewer.so")

root[] new TTreeViewer()

tree viewer

The left panel contains the list of trees and their branches. You can add more trees with the File-Open command to open the file containing the new tree, then use the context menu on the right panel, select SetTreeName and enter the name of the tree to add. On the right are the leaves or variables in the tree. You can double click on any leaf to a histogram it.

The toolbar in the upper part can be used for user commands, changing the drawing option and the histogram name. The lower part contains three picture buttons that draw a histogram, stop the current command, and refresh the tree. The three check buttons toggle the following:

Hist− the histogram drawing mode;

Scan− enables redirecting of TTree::Scan command in an ASCII file;

Rec− enables recording of the last issued command.

To draw more than one dimension you can drag and drop any leaf to the X,Y,Z boxes. Then push the Draw button, witch is marked with the purple icon on the bottom left.

All commands can be interrupted at any time by pressing this button.

The method TTree::Refresh is called by pressing the refresh button in TTreeViewer. It redraws the current exposed expression. Calling TTree::Refresh is useful when a tree is produced by a writer process and concurrently analyzed by one or more readers.

To add a cut/weight to the histogram, enter an expression in the "cut box". The cut box is the one with the scissor icon.

Below them there are two text widgets for specifying the input and output event lists. A Tree Viewer session is made by the list of user-defined expressions and cuts, applying to a specified tree. A session can be saved using File/SaveSource menu or the SaveSource method from the context menu of the right panel. This will create a macro having as default name treeviewer.C that can be ran at any time to reproduce the session.

Besides the list of user-defined expressions, a session may contain a list of RECORDS. A record can be produced in the following way: dragging leaves/expression on X/Y/Z; changing drawing options; clicking the RED button on the bottom when happy with the histogram.

NOTE that just double clicking a leaf will not produce a record: the histogram must be produced when clicking the DRAW button on the bottom-left. The records will appear on the list of records in the bottom right of the tree viewer.

Selecting a record will draw the corresponding histogram. Records can be played using

the arrow buttons near to the record button. When saving the session, the list of records is being saved as well.

Records have a default name corresponding to the Z: Y: X selection, but this can be changed using SetRecordName() method from the right panel context menu. You can create a new expression by right clicking on any of the E() boxes. The expression can be dragged and dropped into any of the boxes (X, Y, Z, Cut, or Scan). To scan one or more variables, drop them into the Scan box, then double click on the box. You can also redirect the result of the scan to a file by checking the Scan box on top.

When the "Rec" box is checked, the Draw and Scan commands are recorded in the history file and echoed on the command line. The "Histogram" text box contains the name of the resulting histogram. By default it is htemp. You can type any name, if the histogram does not exist it will create one. The Option text box contains the list of Draw options. See "Draw Options. You can select the options with the Options menu. The Command box lets you enter any command that you could also enter on the command line. The vertical slider on the far left side can be used to select the minimum and maximum of an event range. The actual start and end index are shown in on the bottom in the status window.

There is an extensive help utility accessible with the Help menu. The IList and OList are to specify an input list of entry indices and a name for the output list respectively. Both need to be of type TList and contain integers of entry indices. These lists are described below in the paragraph "Error! Reference source not found..

## 0.6 Creating and Saving Trees

To create a TTree we use its constructor. Then we design our data layout and add the branches. A tree can be created by giving a name and title: → TTree t("MyTree","Example Tree");

### 0.6.1 Creating a Tree from a Folder Hierarchy

An alternative way to create a tree and organize it is to use folders. You can build a folder structure and create a tree with branches for each of the sub-folders:

TTree folder_tree("MyFolderTree","/MyFolder")

The second argument "/MyFolder"is the top folder, and the "/" signals the TTree constructor that this is a folder not just the title. You fill the tree by placing the data into the folder structure and calling TTree::Fill.

### 0.6.2 Tree and TRef Objects

MyTree→BranchRef();

This call requests the construction of an optional branch supporting table of references (TRefTable). This branch (TBranchRef) will keep all the information needed to find the branches containing referenced objects at each Tree::Fill, the branch numbers containing the referenced objects are saved in the table of references. When the Tree header is saved (via TTree::Write for example), the branch is saved, keeping the information with the pointers to the branches having referenced objects. Enabling this optional table, allow TTree::Draw to automatically load the branches needed to dereference a TRef (or TRefArray) object

### 0.6.3 Autosave

Autosave gives the option to save all branch buffers every n byte. We recommend using Autosave for large acquisitions. If the acquisition fails to complete, you can recover the file and all the contents since the last Autosave. To set the number of bytes between Autosave you can use the TTree::SetAutosave() method. You can also call TTree::Autosave in the acquisition loop every nentry.

### 0.6.4 Trees with Circular Buffers

void TTree::SetCircular(Long64_t maxEntries);

where maxEntries is the maximum number of entries to be kept in the buffers. When the number of entries exceeds this value, the first entries in the Tree are deleted and the buffers used again.

### 0.6.5 Size of TTree in the File

When writing a TTree to a file, if the file size reaches the value stored in the TTree::GetMaxTreeSize(), the current file is closed and a new file is created. If the original file is named "myfile.root, subsequent files are named "myfile_1.root, "myfile_2.root, etc.

Currently, the automatic change of file is restricted to the case where the tree is in the top level directory. The file should not contain sub-directories. Before switching to a new file, the tree header is written to the current file, then the current file is closed. To process the multiple files created by ChangeFile(), one must use a TChain.

The new file name has a suffix "_N where N is equal to fFileNumber+1. By default a Root session starts with fFileNumber=0. One can set fFileNumber to a different value

via TTree::SetFileNumber(). In case a file named "_N already exists, the function will try a file named "__N, then "___N, etc. The maximum tree size can be set via the static function TTree::SetMaxTreeSize(). The default value of fgMaxTreeSize is 1.9 GB. If the current file contains other objects (like TH1 and TTree), these objects are automatically moved to the new file.

### 0.6.6 User Info Attached to a TTree Object

The function TTree::GetUserInfo() allows adding any object defined by a user to the tree that is not depending on the entry number.

tree→GetUserInfo()→Add(myruninfo);

### 0.6.7 Indexing a Tree

Use TTree::BuildIndex() to build an index table using expressions depending on the value in the leaves.

tree→BuildIndex(majorname, minorname);

The index is built in the following way:

a pass on all entries is made like in TTree::Draw()

var1 = majorname

var2 = minorname

sel = $2^{31}$ × majorname + minorname

for each entry in the tree the sel expression is evaluated and the results array is sorted into fIndexValues

Once the index is computed, using the TTree::GetEntryWithIndex(majornumber, minor-

number), one entry can be retrieved.

// to create an index using leaves Run and Event

tree.BuildIndex("Run","Event");

// to read entry corresponding to Run=1234 and Event=56789

tree.GetEntryWithIndex(1234,56789);

Note that majorname and minorname may be expressions using original tree variables e.g.: "run-90000", "event+3*xx"". In case an expression is specified, the equivalent expression must be computed when calling GetEntryWithIndex(). To build an index with only majorname, specify minorname="0" (default).

Once the index is built, it can be saved with the TTree object with:

tree.Write(); //if the file has been open in "update" mode

The most convenient place to create the index is at the end of the filling process just before saving the tree header. If a previous index was computed, it is redefined by this new call.

This function can also be applied to a TChain. The return value is the number of entries in the Index (¡ 0 indicates failure).

## 0.7 Branches

The class for a branch is called TBranch. If two variables are independent, and the designer knows the variables will not be used together, they should be placed on separate branches. If, however, the variables are related, such as the coordinates of a point, it is most efficient to create one branch with both coordinates on it. A variable on a TBranch is called a leaf (TLeaf). Another point to keep in mind when designing trees is that branches

of the same TTree can be written to separate files. To add a TBranch to a TTree we call the method TTree::Branch(). Note that we DO NOT use the TBranch constructor.

The TTree::Branch method has several signatures. The branch type differs by what is stored in it. A branch can hold an entire object, a list of simple variables, contents of a folder, contents of a TList, or an array of objects.

## 0.8 Adding a Branch to Hold a List of Variables

If in the staff.root, the data to save is a list of simple variables, such as integers or floats, we use the following TTree::Branch signature

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

The first parameter is the branch name. The second parameter is the address from which the first variable is to be read.

The leaf name is NOT used to pick the variable out of the structure, but is only used as the name for the leaf. The list of variables needs to be in a structure in the order described in the third parameter. This third parameter is a string describing the leaf list. Each leaf has a name and a type separated by a "/" and it is separated from the next leaf by a ":".

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

The example has two leafs: a floating-point number called temp and an integer named ntrack.

"temp/F:ntrack/I:"

The type can be omitted and if no type is given, the same type as the previous variable is assumed. This leaf list has three integers called ntrack, nseg, and nvtex.

“ntrack/I:nseg:nvtex”

There is one more rule: when no type is given for the very first leaf, it becomes a float (F). This leaf list has three floats called temp, mass, and px.

“temp:mass:px”

The symbols used for the type are:

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

O: [the letter ‘o’, not a zero] a boolean (Bool_t)

The type is used for a byte count to decide how much space to allocate. The variable written is simply the block of bytes starting at the starting address given in the second parameter. It may or may not match the leaf list depending on whether or not the programmer is being careful when choosing the leaf address, name, and type.

By default, a variable will be copied with the number of bytes specified in the type descriptor symbol. However, if the type consists of two characters, the number specifies

the number of bytes to be used when copying the variable to the output buffer. The line below describes ntrack to be written as a 16-bit integer (rather than a 32-bit integer).

“ntrack/I2”

With this 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”);

You can also 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”);

}

## 0.9 Adding a TBranch to Hold an Object

To write a branch to hold an event object, we need to load the definition of the Event class, which is in $ROOTSYS/test/libEvent.so (if it doesnt exist type make in $ROOTSYS/test). An object can be saved in a tree if a ROOT dictionary for its class has been generated and loaded.

root[] .L libEvent.so

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 that will be used as a reference in the TTree::Branch method. Then we create a branch with the TTree::Branch method.

root[] Event *event = new Event()

root[] tree→Branch("EventBranch","Event",&event,32000,99)

To add a branch to hold an object we use the signature above. The first parameter is the name of the branch. The second parameter is the name of the class of the object to be stored. The third parameter is the address of a pointer to the object to be stored. Note that it is an address of a pointer to the object, not just a pointer to the object. The fourth parameter is the buffer size and is by default 32000 bytes. It is the number of bytes of data for that branch to save to a buffer until it is saved to the file. The last parameter is the split-level, which is the topic of the next section. Static class members are not part of an object and thus not written with the object. You could store them separately by collecting these values in a special status object and write it to the file outside of the tree. If it makes sense to store them for each object, make them a regular data member.

### 0.9.1 Setting 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.

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.

When the split-level is 1, an object data member is assigned a branch. If the split-level is 2, the data member objects will be split also, and a split level of 3 its data members objects, will be split. As the split-level increases so does the splitting depth. The ROOT default for the split-level is 99. This means the object will be split to the maximum.

Splitting a branch can quickly generate many branches. Each branch has its own buffer in memory. In case of many branches (say more than 100), you should adjust the buffer size accordingly. A recommended buffer size is 32000 bytes if you have less than 50 branches. Around 16000 bytes if you have less than 100 branches and 4000 bytes if you have more than 500 branches. These numbers are recommended for computers with memory size ranging from 32MB to 256MB. If you have more memory, you should specify larger buffer sizes. However, in this case, do not forget that your file might be used on another machine with a smaller memory configuration.

A split branch is faster to read, but slightly slower to write. The reading is quicker because variables of the same type are stored consecutively and the type does not have to be read each time. It is slower to write because of the large number of buffers as described above. See "Performance Benchmarks" for performance impact of split and non-split mode.

When splitting a branch, variables of different types are handled differently. Here are the rules that apply when splitting a branch.

If a data member is a basic type, it becomes one branch of class TBranchElement.

A data member can be an array of basic types. In this case, one single branch is created for the array.

A data member can be a pointer to an array of basic types. The length can vary, and

must be specified in the comment field of the data member in the class definition. See Input/Output.

Pointer data member are not split, except for pointers to a TClonesArray. The TClonesArray (pointed to) is split if the split level is greater than two. When the split level is one, the TClonesArray is not split.

If a data member is a pointer to an object, a special branch is created. The branch will be filled by calling the class Streamer function to serialize the object into the branch buffer.

If a data member is an object, the data members of this object are split into branches according to the split-level (i.e. split-level ¿ 2).

Base classes are split when the object is split.

Abstract base classes are never split.

All STL containers are supported.

C-structure data members are not supported in split mode.

An object that is not split may be slow to browse.

A STL container that is not split will not be accessible in the browser

### 0.9.2 Exempt a Data Member from Splitting

If you are creating a branch with an object and in general you want the data members to be split, but you want to exempt a data member from the split. You can specify this in the comment field of the data member:

class Event : public TObject { private: EventHeader fEvtHdr; // Don't split the header

### 0.9.3 Adding a Branch to Hold a TClonesArray

ROOT has two classes to manage arrays of objects. The TObjArray can manage objects of different classes, and the TClonesArray that specializes in managing objects of the same class (hence the name Clones Array). TClonesArray takes advantage of the constant size of each element when adding the elements to the array. Instead of allocating memory for each new object as it is added, it reuses the memory. Here is an example of the time a TClonesArray can save over a TObjArray. We have 100,000 events, and each has 10,000 tracks, which gives 1,000,000,000 tracks. If we use a TObjArray for the tracks, we implicitly make a call to new and a corresponding call to delete for each track. The time it takes to make a pair of new/delete calls is about 7 s (10-6). If we multiply the number of tracks by 7 s, (1,000,000,000 * 7 * 10-6) we calculate that the time allocating and freeing memory is about 2 hours. This is the chunk of time saved when a TClonesArray is used rather than a TObjArray. If you do not want to wait 2 hours for your tracks (or equivalent objects), be sure to use a TClonesArray for same-class objects arrays. Branches with TClonesArrays use the same method (TTree::Branch) as any other object described above. If splitting is specified the objects in the TClonesArray are split, not the TClonesArray itself.

### 0.9.4 Identical Branch Names

When a top-level object (say event), has two data members of the same class the sub branches end up with identical names. To distinguish the sub branch we must associate them with the master branch by including a . (a dot) at the end of the master branch name. This will force the name of the sub branch to be master.sub branch instead of

simply sub branch. For example, a tree has two branches Trigger and MuonTrigger, each containing an object of the same class (Trigger). To identify uniquely the sub branches we add the dot:

tree→Branch("Trigger.","Trigger",&b1,8000,1);

tree→Branch("MuonTrigger.","Trigger",&b2,8000,1);

If Trigger has three members, T1, T2, T3, the two instructions above will generate sub branches called: Trigger.T1, Trigger.T2, Trigger.T3, MuonTrigger.T1, MuonTrigger.T2, and MuonTrigger.T3.

## 0.10 Adding a Branch with a Folder

add a branch from a folder

tree→Branch("/aFolder");

This method creates one branch for each element in the folder. The method returns the total number of branches created. "event is a structure with one float and three integers and one unsigned integer. You should not assume that the compiler aligns the elements of a structure without gaps. To avoid alignment problems, you need to use structures with same length members. If your structure does not qualify, you need to create one branch for each element of the structure.

## 0.11 Adding a Branch with a Collection

This Branch method creates one branch for each element in the collection.

tree→Branch(*aCollection, 8000, 99);

// Int_t TTree::Branch(TCollection *list, Int_t bufsize,

// Int_t splitlevel, const char *name)

The method returns the total number of branches created. Each entry in the collection becomes a top level branch if the corresponding class is not a collection. If it is a collection, the entry in the collection becomes in turn top level branches, etc. The split level is decreased by 1 every time a new collection is found. For example if list is a TObjArray*

If splitlevel = 1, one top level branch is created for each element of the TObjArray.

If splitlevel = 2, one top level branch is created for each array element. If one of the array elements is a TCollection, one top level branch will be created for each element of this collection.

In case a collection element is a TClonesArray, the special Tree constructor for TClonesArray is called. The collection itself cannot be a TClonesArray. If name is given, all branch names will be prefixed with name_.

IMPORTANT NOTE1: This function should not be called if splitlevel< 1.

IMPORTANT NOTE2: The branches created by this function will have names corresponding to the collection or object names. It is important to give names to collections to avoid misleading branch names or identical branch names. By default collections have a name equal to the corresponding class name, e.g. the default name of TList is "TList.