1 Preliminaries

A **point cloud** is of type `PointCloud::Dataset` which is effectively a `std::map<std::string, Array>`. An `Array` is contiguous storage with a dynamic numerical type.



A may aggregate many point clouds into a **named point cloud set** is a `std::map<std::string,Dataset>`.



We may also **concatenate** point clouds **by reference** (light weight, no copy) into a **PointCloud::DisjointDataset**.

ref(PC 1_a)	ref(PC 1_b)

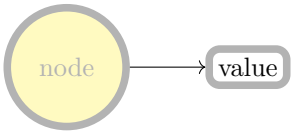
And we may combine the two aggregations `std::map<std::string, DisjointDataset`. In the context of the a **point cloud tree** we will call these **scoped point clouds**.

2 N-ary tree

We now turn to generic *n-ary* trees. A tree is composed of nodes.



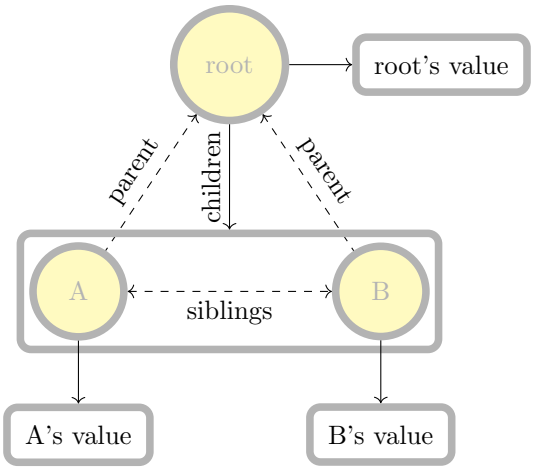
And we will define a node to have **three parts**. First, we allow a node to hold a **value** (aka "payload") object.



Ownership is indicated by the **solid arrow**.

Next, we say a node has **zero or one parent nodes** and a node has **zero or more children nodes**. A tree that requires any node to have no more than two children nodes is a "binary tree". A general or n-ary tree allows "n" children.

We cause a node to **own its children nodes** and a node will **reference its parent node**. The children are organized into an ordered list so children nodes **reference their neighboring siblings**. A tree with three nodes, one parent and two children are illustrated. A reference is indicated with **dashed arrow**. Again, while only two children are drawn, an n-ary tree node may have zero or more.



As implemented here, **a tree is just its nodes**. There is no "tree object" per se. Rather, if code holds a "root" node then that code may access all nodes in the tree. This may involve a manual navigation of the child/parent/sibling edges. But, most conveniently this is done with a **tree descent iteration**.

A **depth-first descent** iteration is implemented. A descent may visit a node and all its descendants. A descent may also be limited to a maximum depth. For example a descent on a root note with a depth of unity (1) will visit only that root node. A descent with a depth of two will visit the root and its direct children. A depth of three will also include grandchildren, etc. The special depth of zero marks unlimited descent.

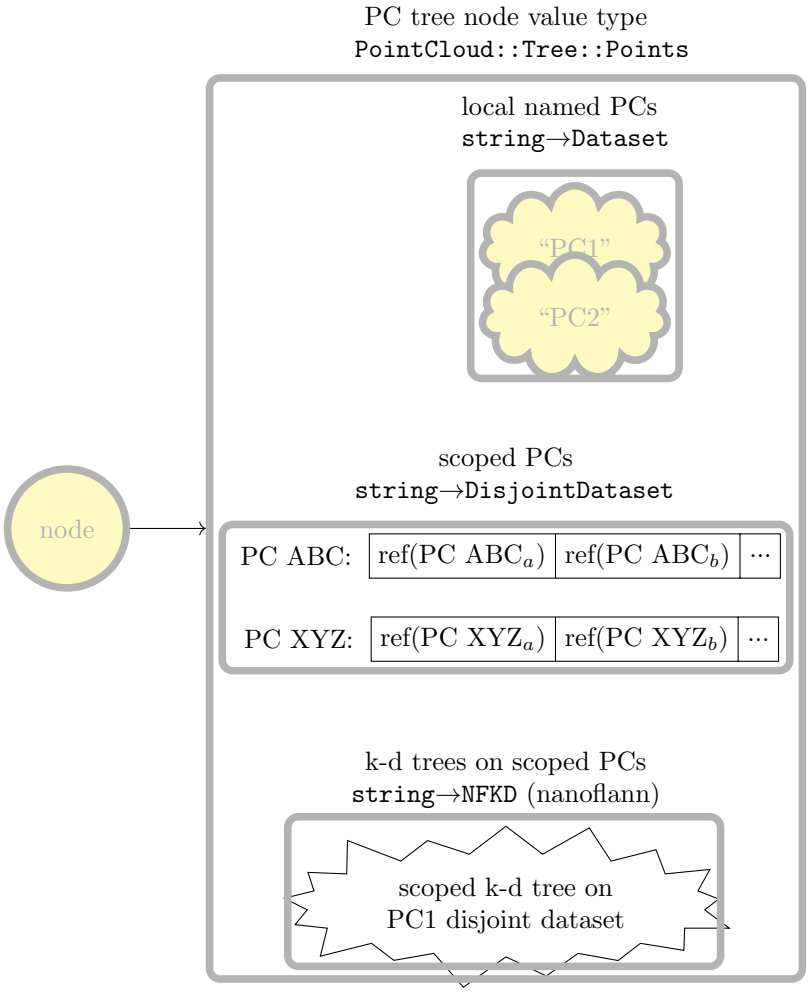
A node with parent may be removed from the tree. The removed node will be removed from its parent's collection of children, its neighbor siblings will themselves become neighbors and the removed node will no longer have a parent. This removed node then becomes a root node of a tree holding its descendants (if any). Conversely, a new node may be added as a child to any node.

3 Point cloud tree

A **point cloud tree** or **point tree** is an n-ary tree with a **node value** type that provides storage and management of **point clouds** and **k-d trees** formed on them. This layer of functionality is provided by a **node value** of type `PointCloud::Tree::Points`. It primarily consists of three parts:

1. a set of "local" named point clouds
2. a set of "scoped" point clouds
3. a set of "scoped" k-d trees built on the scoped PCs

We will described "scoped" below. First, these three parts of the point cloud tree node value are illustrated.



4 Point cloud tree scope

A **scope** is the information that describes a selection of a subset of the data in a point cloud tree. Selected data is said to be "in scope". A scope is defined independently of any tree node but is interpreted respect to a given tree node. This node is considered the "root" of the scope. A scope is defined by three items:

1. A depth-first descent iteration depth as described above.
2. A name of a node-local point cloud.
3. A list of names of point cloud arrays to use as k-d tree coordinates.

A scope may be applied to a point cloud tree to produce a representation of a point cloud called a **scoped point cloud**. A scoped point cloud is represented by a `DisjointDataset` which is a light-weight collection of references to individual point clouds each represented by `Dataset`. The construction of the scoped point cloud follows these rules:

1. A node is provided as the root of the scope.
2. A depth-first descent iteration begins at the root node.
3. The depth of the descent is limited by the scope's `.depth` value.

4. If a visited node lacks a local point cloud with the name given by the scope it is ignored and the iteration continues.

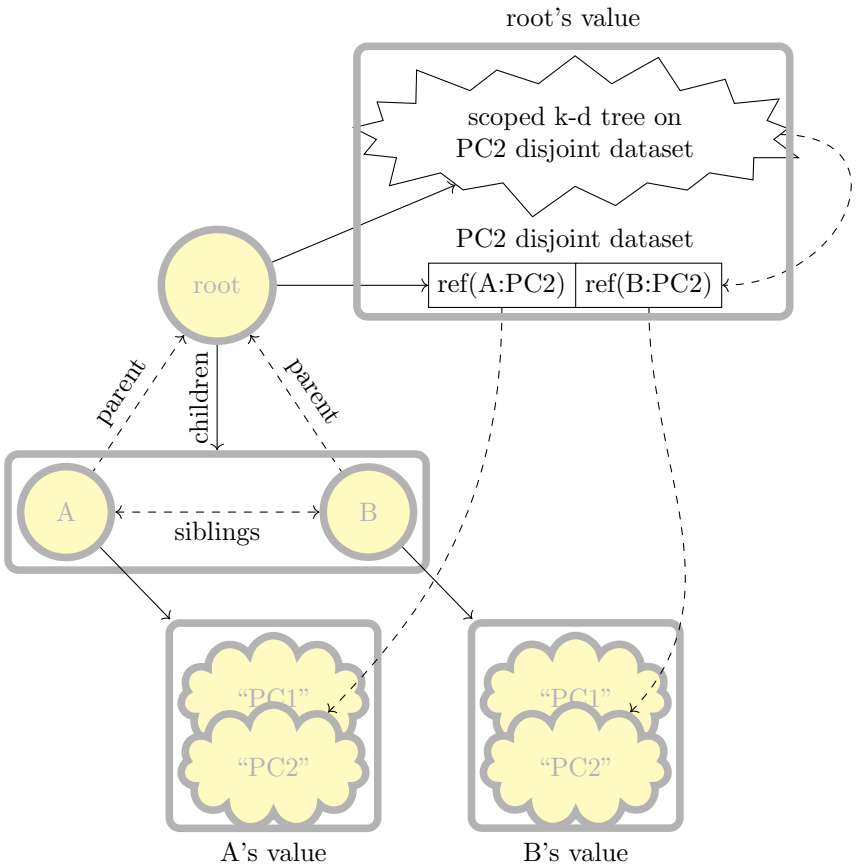
5. If a visited node lacks an array for each coordinate name given in the scope an exception is raised.

6. The node is considered "in scope" and its named local point cloud is appended to the scoped point cloud.

A **scoped k-d tree** is built upon a **scoped point cloud** of the same scope and using the named scope arrays as coordinates. A scoped point cloud may be created without an accompanying scoped k-d tree. Conversely, creating a scoped k-d tree will use an existing or automatically create the corresponding scoped point cloud.

5 Example

A small point cloud tree is illustrated. The "root" node has two children labeled "A" and "B" in the diagram. Each child has a set of point clouds with the names "PC1" and "PC2". The user requests, via the value of the "root" node, a scoped point cloud with a scope name of "PC2", depth not equal to one and a list of point cloud array names (not illustrated). The value of the "root" node performs a depth-first descent iteration collecting "PC2" from children "A" and "B" as they are in scope, construct, stores and returns the lightweight disjoint dataset as the scoped point cloud. The user may later request a k-d tree in the same scope. This will use the previously created scoped point cloud. Had it not been previously constructed it would be made as a side-effect to constructing the scoped k-d tree.



6 Dynamic actions

A point cloud tree supports node insertion and removal with some caveats. For lowest developer cognitive burden, do not modify the tree after requesting any scoped objects. Insertion and removal of nodes may update existing scoped objects to which the caller may hold reference. The caveats are now described.

Insertion is performed such that if the node is in scope for any existing scoped point clouds or scoped k-d trees these objects will be updated. While the initial construction of a scoped point cloud reflects the order of the depth-first descent iteration a later update will effectively append the new local point cloud of the inserted node to the existing scoped point cloud and/or scoped k-d tree.

Removal is less sophisticated and stable. Removing a node that is in scope for existing scoped point clouds or scoped k-d trees will invalidate these objects. The invalidation is lazy. That is, any existing handles on these objects will not reflect the removal. If the scoped object is retrieved again from the node or if an in-scope descendant node is added the scoped object is rebuilt. The order of points in the scoped object will reflect a fresh depth-first descent. Any references to any point ordering held by the caller must be considered invalid.