What is the correct structure for a node in any BST Tree

Asked 9 years, 6 months ago Modified 7 years, 4 months ago Viewed 1k times

2

What is the correct way based on the theory to create a Node for a Binary Tree? For example:

```
struct Node
{
    int data;
    Node *left;
    Node *right;
};
```

The problem I'm currently facing is that I have 2 different answers from several sources (books,website,online lectures.. etc).

From "Introduction to Algorithms",edition 3, p 286,287 : "In addition to a key and satellite data, each node contains attributes left, right, and p that point to the nodes corresponding to its left child,its right child, and its parent, respectively."

Which means something like this:

```
struct Node
{
   int data;
   Node *parent;
   Node *left;
   Node *right;
};
```

On the other hand, I found several links which DO NOT follow this design such as:

http://algs4.cs.princeton.edu/32bst/

http://math.hws.edu/eck/cs225/s03/binary_trees/

http://www.cprogramming.com/tutorial/lesson18.html

These implementations DO NOT keep a link to the parent and from some online lectures it is said that Trees do NOT traverse backwards (aka. can't see the parent) which counters the notion from the book!

In RedBlack trees for instances you NEED to see the grandparent and uncle of that node to determine whether to re-colour and/or rotate to rebalance the tree. In AVL trees you don't since the focus is on the height of sub-trees. Quad Trees and Octrees are the same that you don't need the parent.

Questions:

Can someone please answer me this and with valid sources explain which is the CORRECT way to design a node for a Binary Tree or for Any Tree (B-Trees,..etc)?

Also what is the rule with Traversing Backwards? I know of Pre-order, In-order, Post-order, Breadth-First, Depth-First(Pre-order) and other AI Heuristic algorithms for traversals.

Is it true that you are NOT allowed to move backwards in a tree ie from child to parent? If so, then why does the book suggest a link to parent node?

c++ algorithm binary-search-tree

Share
Improve this question
Follow

edited Jun 25, 2015 at 14:39
Kara
6,206 ● 16 ● 53 ● 58

asked Jun 23, 2015 at 17:49
Constantinos Glynos
3,172 ● 2 ● 17 ● 33

Generally it is the first case, however it depends on your implementation – Ediac Jun 23, 2015 at 17:51

The pointer to the parent makes implementation of *iterators* easier; may also remove the need for recursion or a stack when traversing. – Thomas Matthews Jun 23, 2015 at 18:00

What do you mean by backwards traversal? Remember that we generally use stacks for traversals (recursion uses the program stack) so to change traversal type, you just change where you put the visiting code. – scohe001 Jun 23, 2015 at 18:00

6 Answers

Sorted by: Highest score (default)

5

The fundamental Binary Tree (foundation) requires child pointers:

```
struct binary_tree_node
{
  binary_tree_node * left_child;
  binary_tree_node * right_child;
};
```

There are many modifications that can be made to the foundation that help facilitate searching or storage.

These can include (but are not limited to):

- parent pointer
- array of child pointers
- "color" indicator
- specialized leaf nodes -- no child links

The amenities depend on the usage of the data structure. For example, an array of child nodes may help speed up I/O access, where reading a "page" node is as efficient as reading a single node (See B-Tree). The "color" indicator may help with the decision for balancing. The specialized "leaf" nodes reduce the amount of memory occupied by the tree.

As for traversal, a tree can be traversed in any method. There are no rules preventing a traversal from child to parent. Some traversals may include sibling to sibling.

Some books or websites may pick nits about a traditional or fundamental "binary tree" data structure. I find that restrictions get in the way.

Share Improve this answer Follow

answered Jun 23, 2015 at 18:28

Thomas Matthews

57.6k ● 17 ● 105 ● 163

Right.. So it's just a matter of preference based on implementation. Obviously having an extra pointer increases the size of the node but if it means you can access ancestors easily instead of re-traversing or keeping local pointers to parents, then I agree it makes sense to add the extra pointer in the Node. My confusion arose from the different reading material not following the same rule. But I think your answer cleared it up a bit. – Constantinos Glynos Jun 24, 2015 at 8:38

1

There isn't any canonical definition.

In general, imperative-language (e.g., C++) tend to favor the with-parent approach. It simplifies the implementation of efficient rebalancing, and, as Thomas Matthews pointed out, facilitates constant-space iterators.

Functional languages (e.g., Haskell), tend to use the no-parent approach (see [Purely Functional Data Structures]). Since no modifications are possible, all rebalancing is done by recopying along the search path anyway, so no back pointer is needed.

Being strongly recursion oriented, the design of a constant space iterator is also not much of a concern there.

Share Improve this answer Follow

answered Jun 23, 2015 at 18:05

Ami Tavory
76.2k ● 13 ● 148 ● 193

1

There is no hard and fast rule that there must be a link back to the parent in your tree data structure. Having a link back to the parent is analogous to a doubly linked list. Not having a link back to the parent is just a linked list. With a back link, obviously you gain more flexibility, but at the expense of (relatively) more complicated implementation. Many problems can be solved with a linked list while some others require a doubly linked list.

Share
Improve this answer
Follow

edited Jun 23, 2015 at 18:06

answered Jun 23, 2015 at 18:02

Kevin Le - Khnle
10.9k ● 11 ● 56 ● 84

A parent link also takes up more space, each node is bigger, including the leaf nodes.
– Thomas Matthews Jun 23, 2015 at 18:03

1

It depends on your task

Truly speaking `binary search tree` is a concept and there is no strict or standard rules for designing the data structure. But to understand the basic functionality (eg. insert, delete, find etc.) people use very basic data structure like,

```
struct Node
{
    int data;
    Node *left;
    Node *right;
};
```

But it is your task which may design it differently for different purpose. For example, given a `tree node` at some point of your task if you need to find its `parent node` in single operation you might think to design the node struct like,

```
struct Node
{
    int data;
    Node *parent;
    Node *left;
    Node *right;
};
```

Some other complex implementations may require to store a list of siblings too. Which will be like,

```
struct Node
{
    int data;
    Node *parent;
    Node *left;
    Node *right;
    list<Node> *siblings;
};
```

So, there is no strict standard

Share Improve this answer Follow

answered Dec 27, 2015 at 16:34
Sazzad Hissain Khan
40k ● 41 ● 208 ● 298

-1

```
struct tree_node
{
  tree_node* left_child;
  tree_node* right_child;
  int data;  // here you can use whatever type or data you want. Even generic
type
};
```

Share Improve this answer Follow

answered Jun 23, 2015 at 19:55
BufBills
8,083 ● 14 ● 49 ● 94

-2

The following node definition (in Java) is for a balanced binary tree rather than a BST.

```
// Copyright (C) NNcNannara 2017

public class Node
{
    public Node Left;
    public Node Right;
    public Node Parent;
    public State Balance;

    public Node()
    {
        Left = this;
        Right = this;
        Parent = null;
        Balance = State.Header;
    }

    public Node(Node p)
    {
        Left = null;
        Right = null;
        Parent = p;
        Balance = State.Balanced;
    }

    public Boolean isHeader ()
    { return Balance == State.Header;  }
}
```

This is optimised for balancing routines. The idea that a set node derives from Node as follows.

```
// Copyright (C) NNcNannara 2017

public class SetNode<T> extends Node
{
    public T Data;

    public SetNode(T dataType, Node Parent)
    {
        super(Parent);
        Data = dataType;
    }
}
```

And a dictionary node is as follows.

```
// Copyright (C) NNcNannara 2017

public class DictionaryNode<K, T> extends Node
{
    public T Data;
    public K Key;

    public DictionaryNode(K keyType, T dataType, Node Parent)
    {
        super(Parent);
        Key = keyType;
        Data = dataType;
    }
}
```

Balancing and iteration are non-generic in nature and are defined for the base class Node. Of course, binary trees may also exist on disk, whereby the node type is as follows.

```
package persistent;

public class Node
{
    public long Left;
    public long Right;
    public long Parent;
    public long Key;
    public calculus.State Balance;

    public Node()
    {
        Left = 0;
        Right = 0;
        Parent = 0;
        Balance = calculus.State.Header;
        Key = 0;
    }

    public Node(long p)
    {
        Left = 0;
        Right = 0;
        Parent = p;
        Balance = calculus.State.Balanced;
        Key = 0;
    }

    public Boolean IsHeader ()  { return Balance == calculus.State.Header; }
}
```

Instead of references being present, long integer offsets into the node and data file are present. Note that there is only one node type for all collections on disk.

Share Improve this answer Follow

answered Aug 16, 2017 at 11:29

Benedict NNcNannara

1 ● 3

See rosettacode.org/wiki/AVL_tree#Generic for the full source code to AVL Trees in the Generic Language (gigasopht.net). – Benedict NNcNannara Nov 12, 2021 at 10:29