TREE

Project ☺

Defind New Data Structure!!

If we can put the data in the middle of an array, then we can do insert_first and delete_first in constant time, but programming languages such as Python and cpp are not able to allocate free space before the beginning of the data, and the navigation is always from the side. Left to right.

Define a data structure that addresses this issue

Things you need to know:

- Array
- Linked List
- Dynamic Array

Our Goal

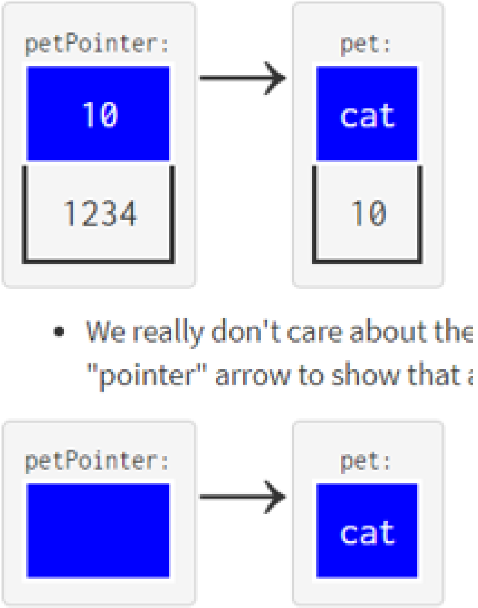
Data Structure	Build()	Get_at(i)/ Ste_at(i, x)	Insert_at(i, x) / Delete_at(i)
Array	n	n	n
Linked List	n	1	n
Goal	n	$\log n$	$\log n$

How? Binary Trees!

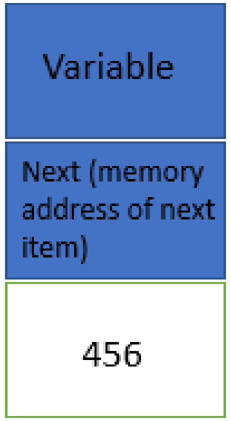
- Pointer-based data structures (like Linked List) can achieve worst-case performance
- Binary tree is pointer-based data structure with three pointers
- per node Node representation: node.{item, parent, left, right}

Remember Pointer

Remember how we understand pointers

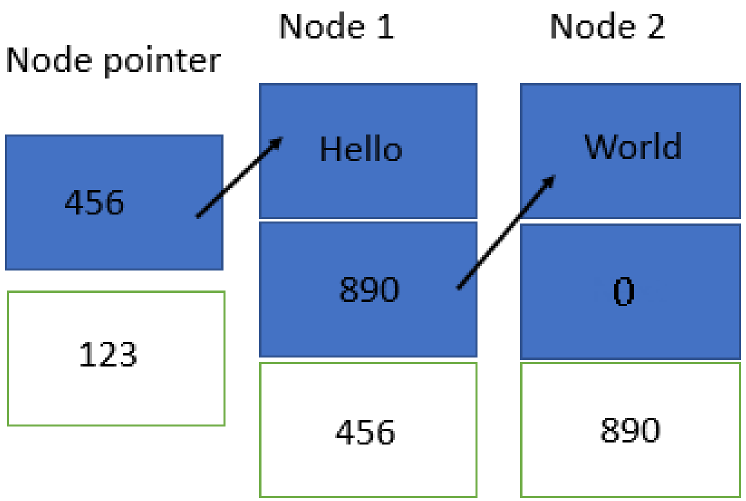


A node is a struct that combines the principles of a pointer and variable. It's a pointer with a variable

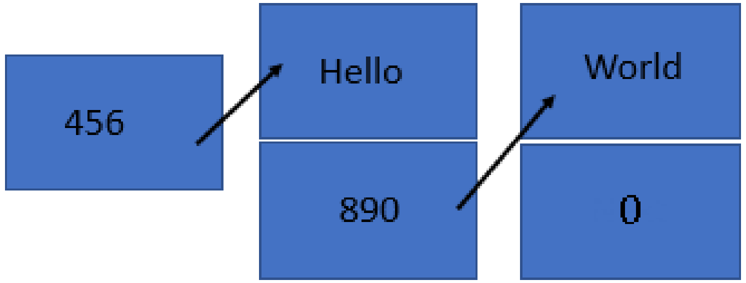


4 boxes become three because memory address is the same

A linked list



Remember we said we don't really care about the "memory address" box so we can omit it for our shorthand



One Parent, No Cycles

example

_____________<A>__________

<B> _______<C>_______

<D>______ <E>

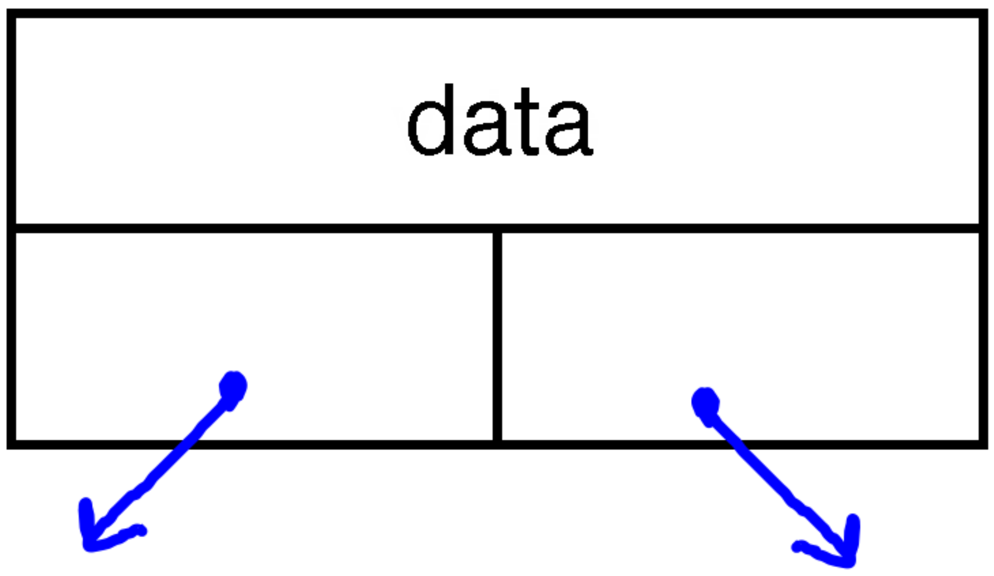
<F>

Node	<A>	<B>	<C>	<D>	<E>	<F>
Item	A	B	C	D	E	F
Parent	_	<A>	<A>	<C>	<C>	<D>
Left	<B>	_	<D>	_	_	_
right	<C>	_	<E>	<F>	_	_

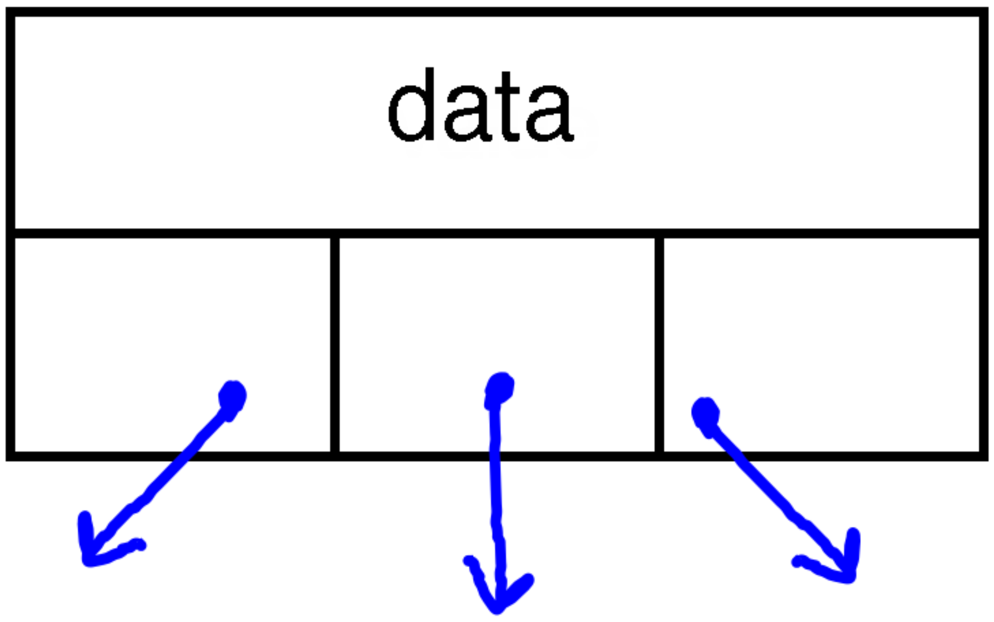
Building trees programatically

- Each node must have data value
- Each node must have left child pointer
- Each node must have right child pointer
- Each node must have parent pointer

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
};
```



```
struct TernaryTreeNode {
    string data;
    TernaryTreeNode* left;
    TernaryTreeNode* middle;
    TernaryTreeNode* right;
};
```



attribute

- Node: node->left child -> parent = node
- Subtree(X): X and descendants (X as root)
- Depth(X): # edges in path from X to the root
- Hight(X): # edges in longest downward path from X (max depth in subtree(X)

Tree Traversal Techniques

- **Preorder Traversal:** Visit_ Left_ Right
- **Inorder Traversal:** Left_ Visit_ Right
- **Postorder Traversal:** Right_ Visit_ Left