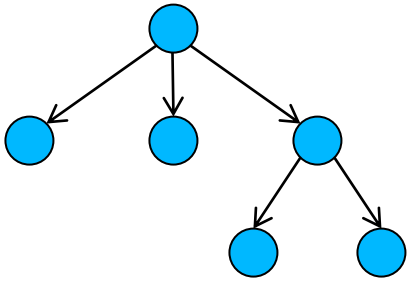
BINARY TREES

OVERVIEW

OVERVIEW

- **What is a tree?**

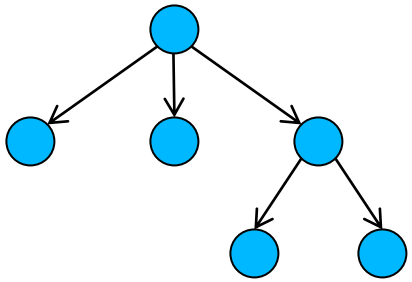
Good for climbing



Good for storing data

OVERVIEW

- **In computer science a tree is an abstract data type that stores data in a hierarchical way, much like a family tree**



  - For tree node has most one parent
  - Each tree node has zero or more children
  - Each tree node has zero or more siblings

OVERVIEW

- **Different types of data can be stored in tree nodes depending on needs of the application**
  - Numbers, characters, strings
  - Objects, other ADTs

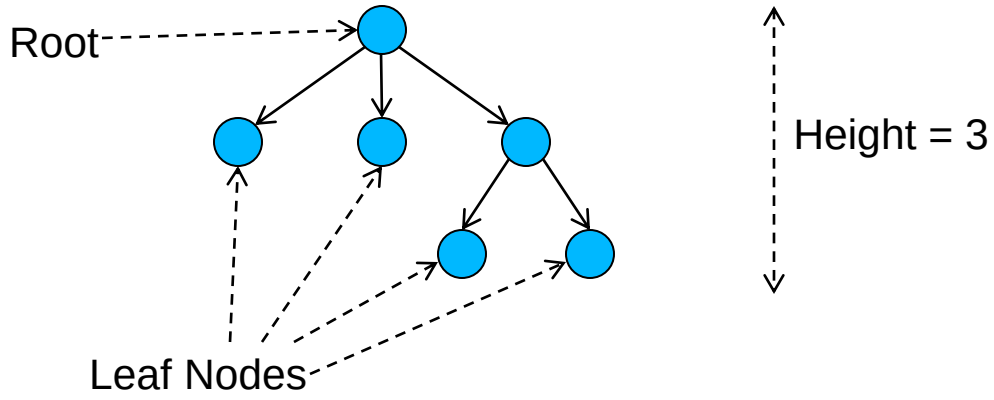
- **When we limit the number of children each node can have to two, we have a binary tree**
  - Useful for quickly storing and retrieving data
  - Also used to represent arithmetic expressions

BINARY TREES

TREE TERMINOLOGY

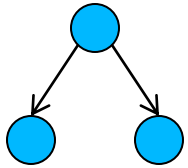
TREE TERMINOLOGY

- **Root – top of tree, has no parent**
- **Leaf – bottom of tree, has no children**
- **Internal – not root, not leaf**
- **Height – the number of nodes on the longest path from leaf node to root node**

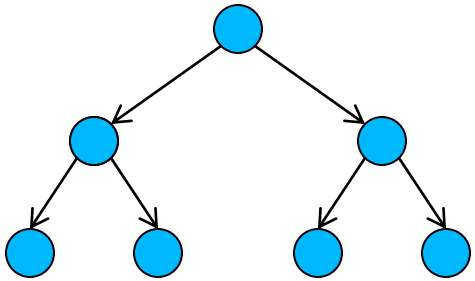


TREE TERMINOLOGY

- **Empty – tree with zero nodes**
- **Full – binary tree with all leaf nodes at level h and all other nodes have 2 children**



Full tree, height 2



Full tree, height 3

TREE TERMINOLOGY

- **How many nodes N can we store in a <u>full</u> binary tree of height h?**

h = 1, N = 1

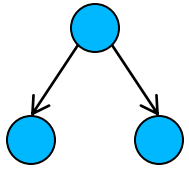
h = 2, N = 1+2 = 3

h = 3, N = 1+2+4 = 7

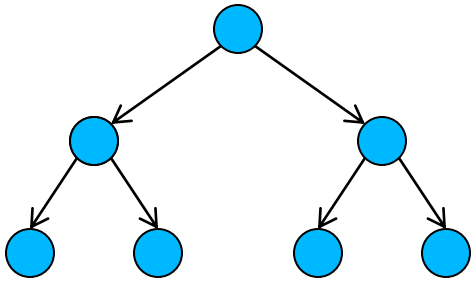
h = 4, N = 1+2+4+8 = 15

…

$N = 1+2+\ldots+2^{h-1} = 2^h-1$



h=2, $N= 2^2-1=3$



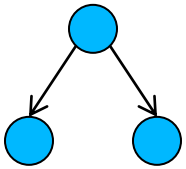
h=3, $N= 2^3-1=7$

TREE TERMINOLOGY

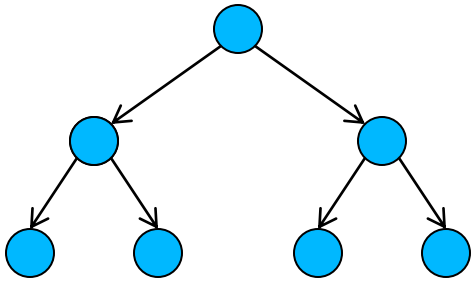
- **What is the <u>minimum</u> height of a binary tree that contains N nodes? Assume tree is full.**

$$N = 2^h - 1$$

$$h = \log_2(N+1)$$



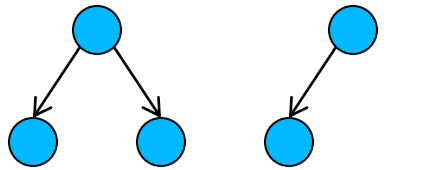
N=3, $h=\log_2 4 = 2$



N=7, $h=\log_2 8 = 3$

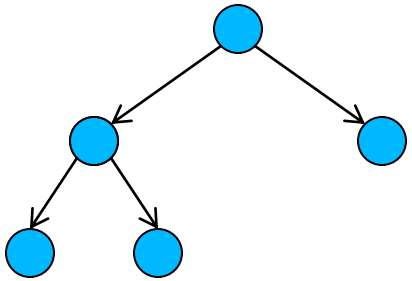
TREE TERMINOLOGY

- **Complete – a binary tree that is full to level h-1 and all leaf nodes on level h are filled in from left to right**

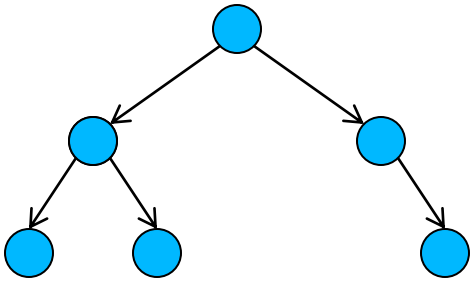


Complete

Complete



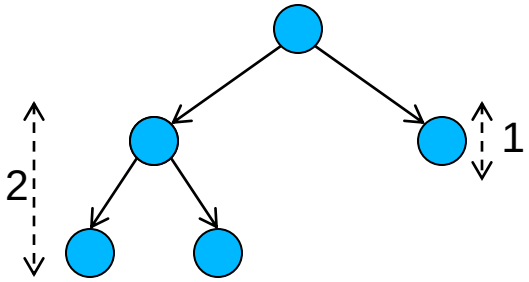
Complete



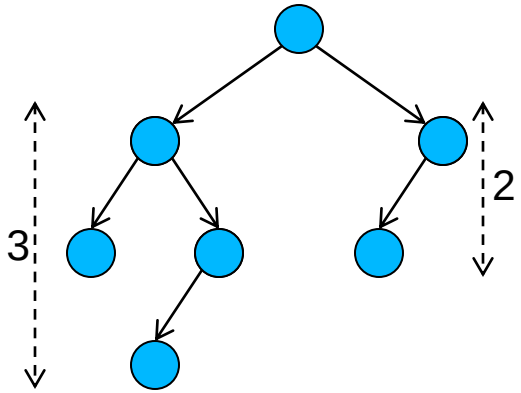
Not complete

TREE TERMINOLOGY

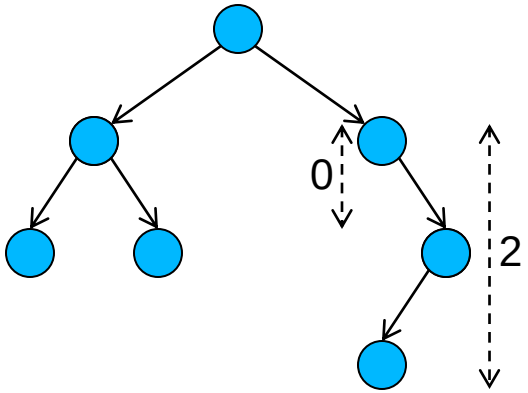
- **Balanced – a binary tree in which the height of the left and right subtrees of any node in the tree differ by at most one**



Balanced



Balanced



Not balanced

BINARY TREES

BINARY SEARCH TREES

BINARY SEARCH TREES

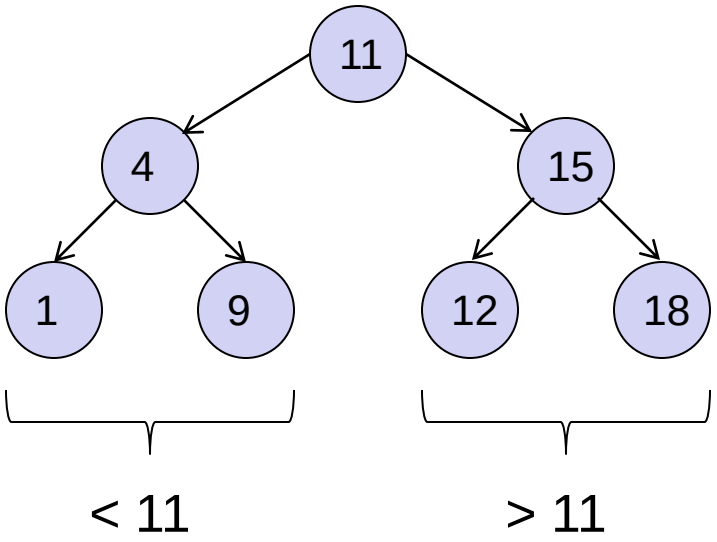
- **Consider the task of searching a sorted array of data using binary search**

1	4	9	11	12	15	18
0	1	2	3	4	5	6

  - We will always look at at data[3]=11 first
  - If value is < 11 we will look at data[1]=4 next
  - If value is > 11 we will look at data[5]=15 next
  - This continues until we find the desired value

BINARY SEARCH TREES

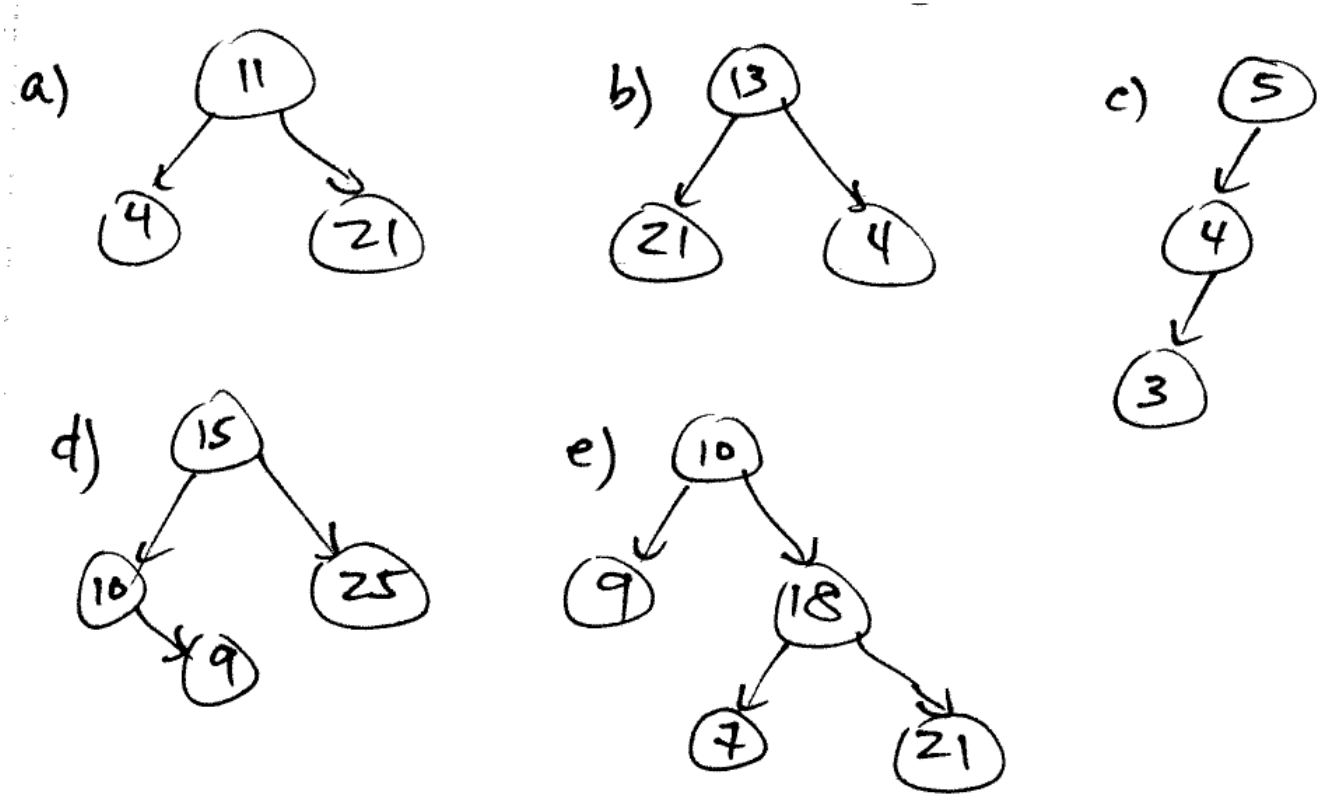
- **This sequence of decisions can be stored in a binary search tree (BST)**



- All nodes in the left subtree are smaller in value than parent
- All nodes in the right subtree are larger in value than parent
- This is true for all nodes in BST

BINARY SEARCH TREES

Which of the following are valid BSTs?

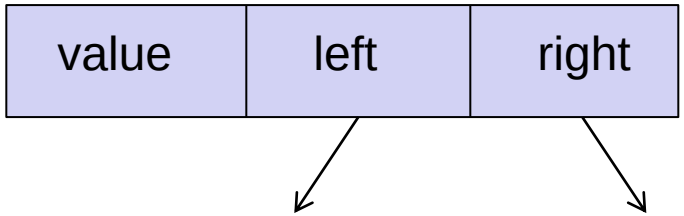


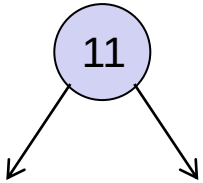
BINARY TREES

BST CLASS DEFINITION

BST CLASS DEFINITION

```
class node
{
public:
    int value;
    node *left;
    node *right;
};
```





BST CLASS DEFINITION

```
class BST
{
public:
        BST();
        ~BST();

        // public methods
        void print();
        bool search(int value);
        bool insert(int value);
        bool delete(int value);
...
```

BST CLASS DEFINITION

```
...
private:
        // private methods
        void print(node* ptr);
        bool search(node* ptr, int value);
        bool insert(node* &ptr, int value);
        bool delete(node* &ptr, int value);
        bool delete_node(node* &ptr);

        // pointer to root of tree
        node *root;
};
```

BINARY TREES

BST PRINT

BST PRINT

- **Assume you are given a valid BST and you want to print all the values in the tree**

- **With an array or linked list, we can start at one end and use a loop access all the data up to the other end**

- **In order to print a BST, we can not simply loop over the nodes. We need to call a recursive function that visits all of the nodes in the tree**
  - We need to pass in a pointer to the root of tree
  - Make recursive calls with left and right pointers
  - The order we visit nodes determines print order

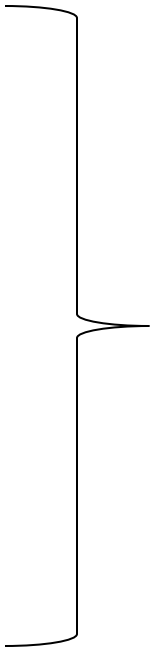
BST PRINT

```
void BST::print1(node* ptr)
{       // terminating condition
        if (ptr == NULL) return;

        // print left subtree
        print1(ptr->left);

        // print node value
        cout << ptr->value << endl;

        // print right subtree
        print1(ptr->right);
}
```

This will print the data values in sorted order

BST PRINT

```
void BST::print2(node* ptr)
{       // terminating condition
        if (ptr == NULL) return;

        // print node value
        cout << ptr->value << endl;

        // print left subtree
        print2(ptr->left);

        // print right subtree
        print2(ptr->right);
}
```



This will print the data values in preorder

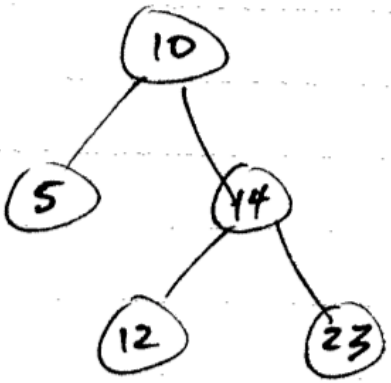
BST PRINT

```
void BST::print3(node* ptr)
{       // terminating condition
        if (ptr == NULL) return;

        // print left subtree
        print3(ptr->left);

        // print right subtree
        print3(ptr->right);

        // print node value
        cout << ptr->value << endl;
}
```

This will print the data values in postorder

BST PRINT

Example with numerical data:



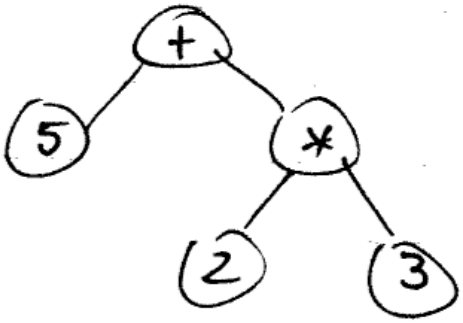
inorder: 5 10 12 14 23

preorder: 10 5 14 12 23

postorder: 5 12 23 14 10

BST PRINT

Example with symbolic data:



inorder: 5 + 2 * 3

preorder: + 5 * 2 3

postorder: 5 2 3 * +

BST PRINT

- **The print functions above have a node* ptr parameter**
  - The user of the BST class should never have direct access to the private root pointer
  - For this reason, we call the private print method from the public print method as follows

```
void BST::print()
{   // call private method
   print1(root);
}
```

BST PRINT

- **How many steps will this print function take?**
  - We have to visit all N nodes, so print takes O(N) steps
  - This is one place where we do not need to worry about how balanced or unbalanced the tree is

- **Is it possible to print a tree using iteration?**
  - Yes, but it is much more complicated
  - We need to use a stack to keep track of nodes to print
  - This is why recursion is so important to master

BINARY TREES

BST SEARCH

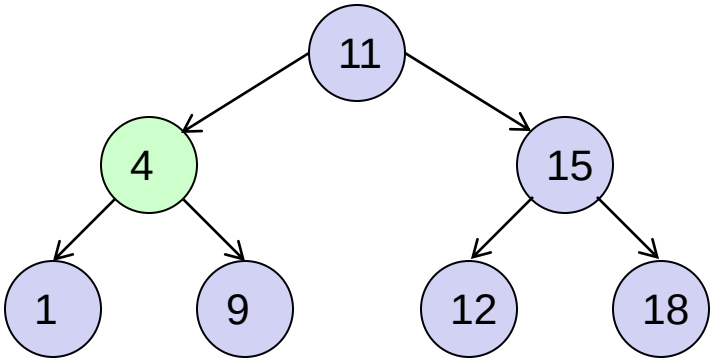
BST SEARCH

- **Assume we are given a valid BST and wish to locate a desired value in the tree**

- **Search Algorithm:**
  - Start ptr at root of tree
  - If node value > desired go to left child
  - If node value < desired go to right child
  - Stop when ptr is null or when value is found

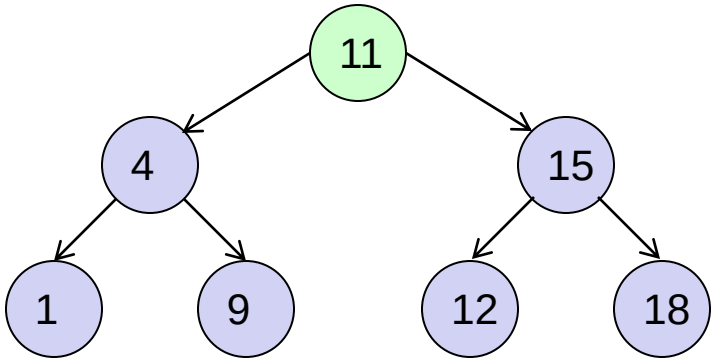
BST SEARCH

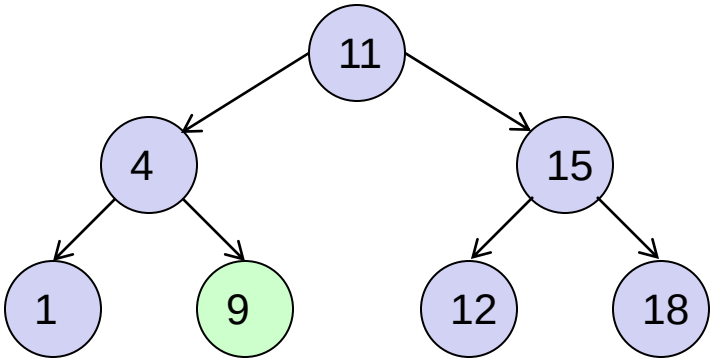
- **Assume we are searching the BST for the value 9**



start at root of tree

9 > 4 so go right



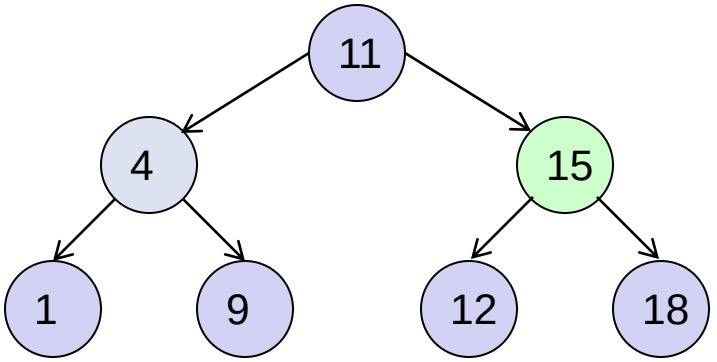


9 < 11 so go left

we found the 9 node

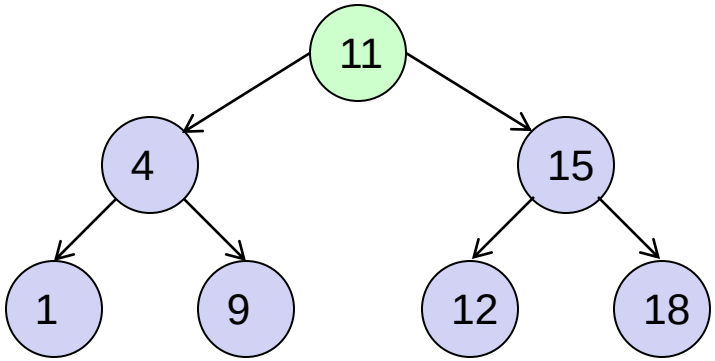
BST SEARCH

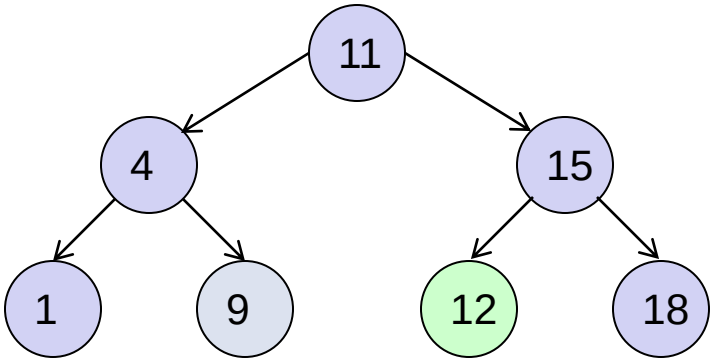
- **Assume we are searching the BST for the value 13**



start at root of tree

13 < 15 so go left





13 > 11 so go right

13 > 12 but null pointer to right so the value not found

BST SEARCH

```
bool List::search(int value)
{
        // iteratively search linked list
        node *ptr = head;
        while ((ptr != NULL) && (ptr->value != value))
        {
                // go to next node
                ptr = ptr->next;
        }
        // return true/false if found or not
        return((ptr != NULL) && (ptr->value == value));
}
```

BST SEARCH

```
bool BST::search1(node* ptr, int value)
{       // iteratively search tree
        while ((ptr != NULL) && (ptr->value != value))
        {       // search left or right subtree
                if (ptr->value > value)
                        ptr = ptr->left;
                else if (ptr->value < value)
                        ptr = ptr->right;
        }
        // return true/false if found or not
        return((ptr != NULL) && (ptr->value == value));
}
```

BST SEARCH

```
bool BST::search2(node* ptr, int value)
{       // terminating conditions
        if (ptr == NULL)
                return false;
        else if (ptr->value == value)
                return true;

        // recursively search tree
        if (ptr->value > value)
                return search(ptr->left, value);
        else if (ptr->value < value)
                return search(ptr->right, value);
}
```

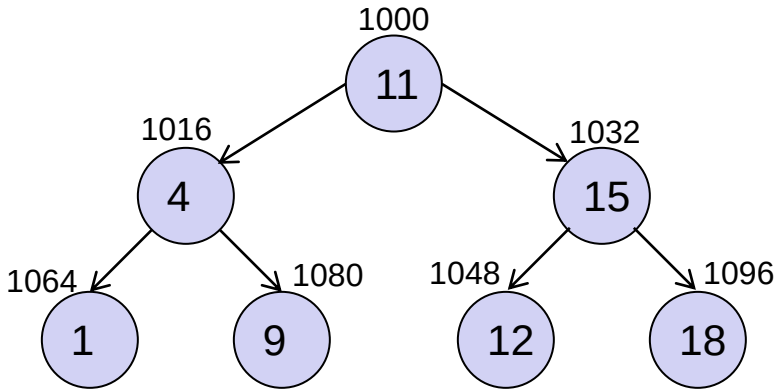
BST SEARCH

- **The search functions above have a node *ptr parameter**
  - The user of the BST class should never have direct access to the private root pointer
  - For this reason, we call the private search method from the public search method as follows

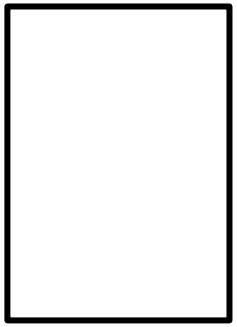
```
bool BST::search(int value)
{   // call private method
   return search2(root, value);
}
```

BST SEARCH

- **Assume we have a BST with nodes at these memory locations**
- **Lets do the box method trace of the call to tree.search(9)**

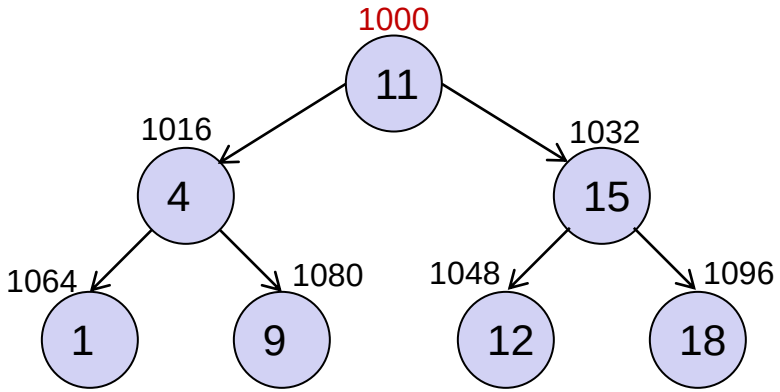


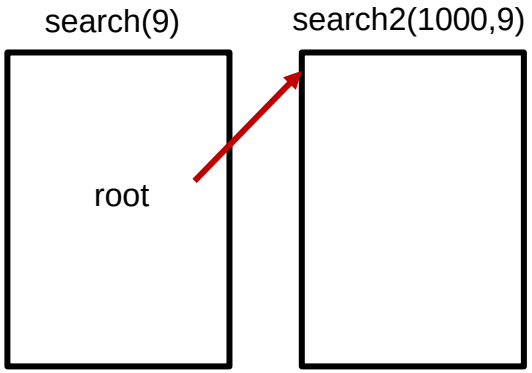
search(9)



BST SEARCH

- **The root of the BST is at memory location 1000 so we call tree.search2(1000, 9)**



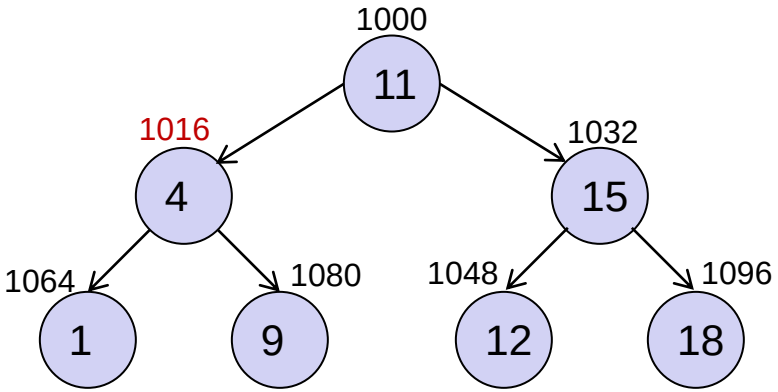


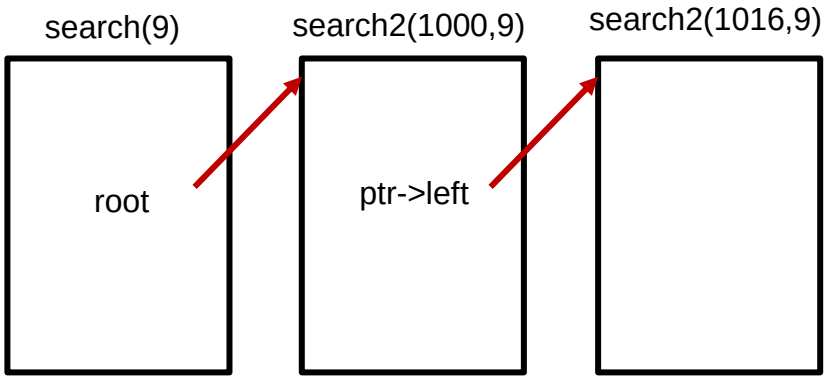
BST SEARCH

- **ptr->value is 11 > 9 so**

we search left and call call

tree.search2(1016, 9)



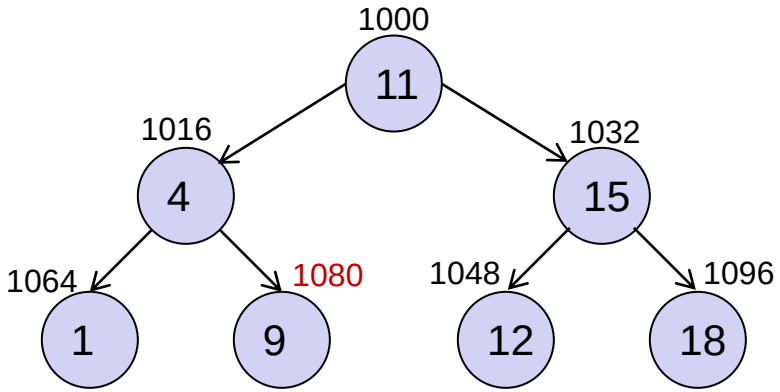


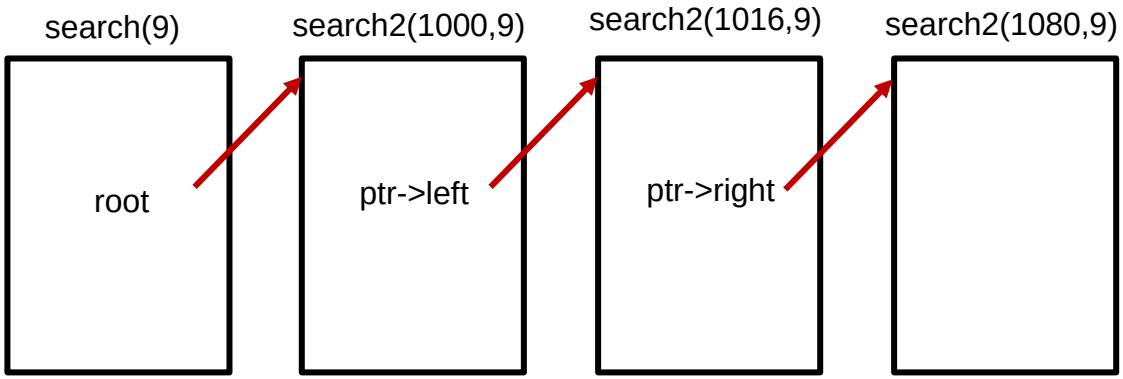
BST SEARCH

- **ptr->value is 4 < 9 so**

we search right and call call

tree.search2(1080, 9)



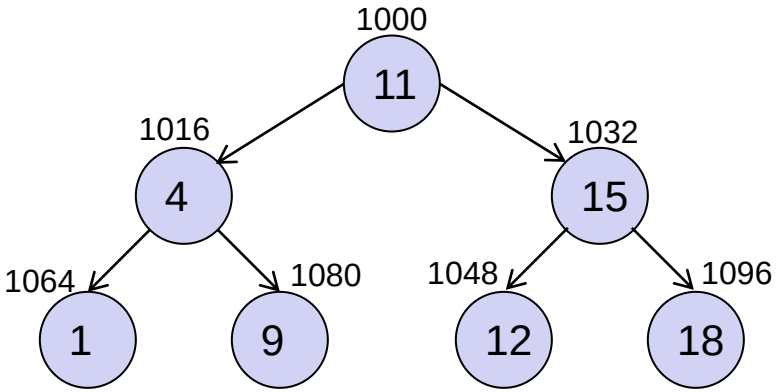


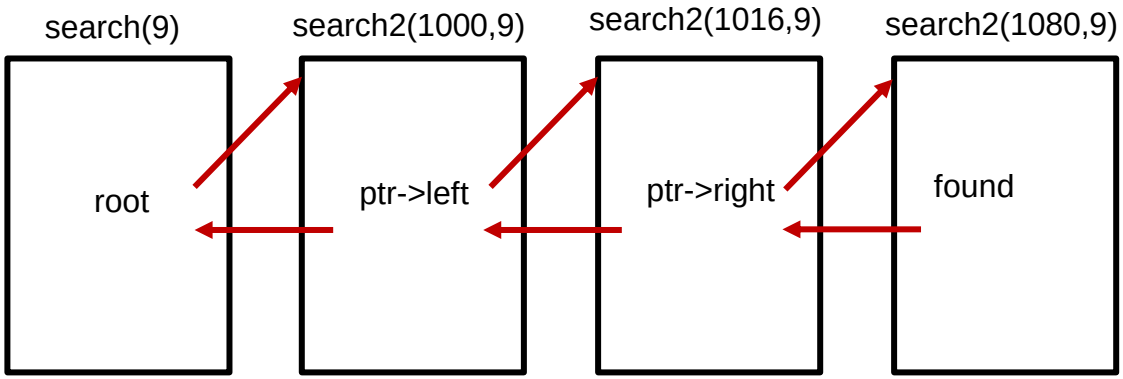
BST SEARCH

- **ptr->value is 9 == 9 so**

we return true all the way back

to the search(9) call





BST SEARCH

- **If we have a balanced tree, this BST search algorithm will find the data after $O(\log_2 N)$ steps**
- **If we have a very unbalanced tree (that looks like a linked list) this BST search may take O(N) steps**
- **On average, we can expect BST search to take $O(\log_2 N)$ steps**

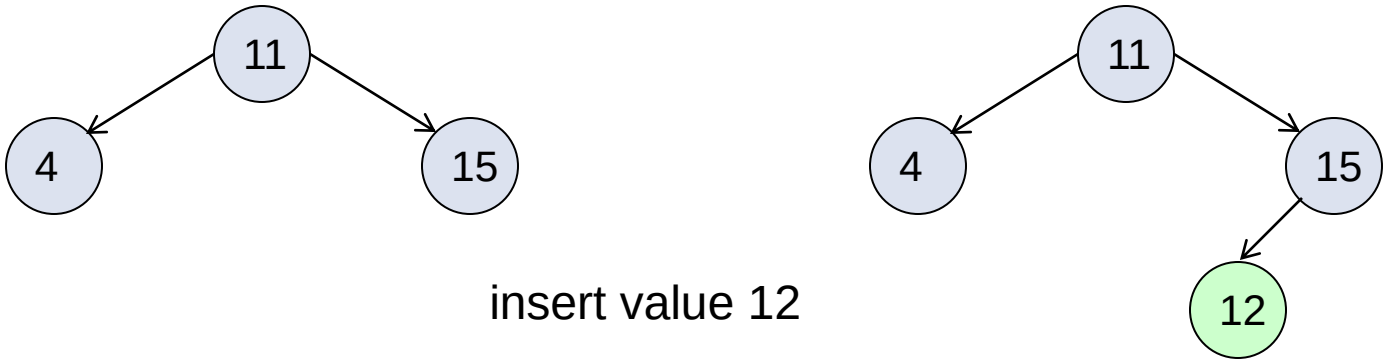
BINARY TREES

BST INSERT

BST INSERT

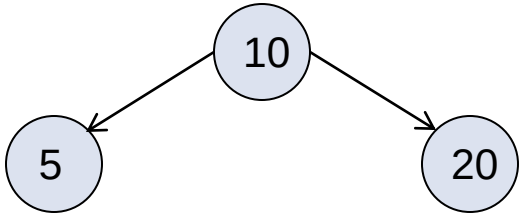
- **Assume we want to insert a new data value into a BST**
  - We want to make sure we will still have a valid BST after insertion so we must insert data where we expect to find it

- **Insert Algorithm:**
  - Search the BST for the desired value
  - Add the new node at the “dead end”

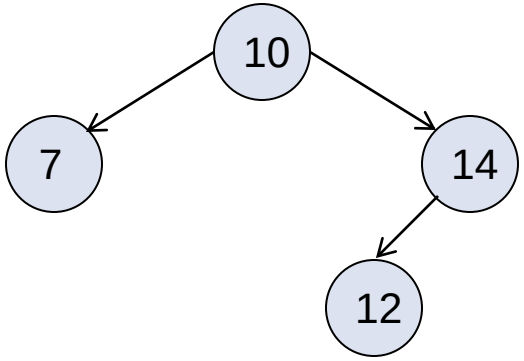


BST INSERT

Examples:



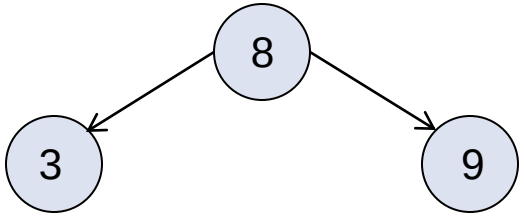
insert 2 and 8



insert 13 and 14

BST INSERT

Examples:



insert 10, 11, and 12
insert 12, 10, and 11

empty tree

insert 42

BST INSERT

- **With this algorithm we are always inserting a new leaf node (never an internal node)**

- **Order matters when inserting values into a BST**
  - What will happen if we insert N sorted values into a BST?
  - What will happen if we insert N random values into a BST?

BST INSERT

```
bool BST::insert(node* ptr, int value)
{
        // terminating condition
        if (ptr == NULL)
        {
                // insert node into BST
                ptr = new node;
                ptr->value = value;
                ptr->left = NULL;
                ptr->right = NULL;
                return true;
        }
```

…

BST INSERT

```
…
        // recursive search and insert
        else if (ptr->value > value)
                return insert(ptr->left, value);
        else if (ptr->value < value)
                return insert(ptr->right, value);
       return false;
}
```

- Do you see any problems with function parameters?
- What will this function do if we insert duplicate data?

BST INSERT

```
bool BST::insert(node* &ptr, int value)
{
        // terminating condition
        if (ptr == NULL)
        {
                // insert node into BST
                ptr = new node;
                ptr->value = value;
                ptr->left = NULL;
                ptr->right = NULL;
                return true;
        }
```

…

BST INSERT

```
...
        // recursive search and insert
        else if (ptr->value >= value)
                return insert(ptr->left, value);
        else if (ptr->value < value)
                return insert(ptr->right, value);
}
```

- **This will insert duplicate values into left subtree.**

BST INSERT

```
...
        // recursive search and insert
        else if (ptr->value > value)
                return insert(ptr->left, value);
        else if (ptr->value <= value)
                return insert(ptr->right, value);
}
```

- **This will insert duplicate values into right subtree.**

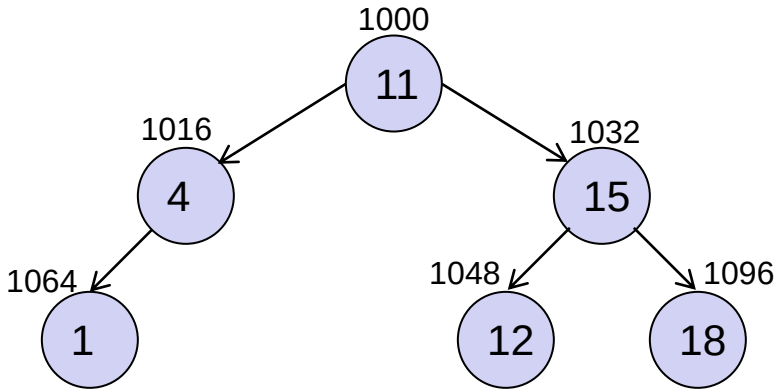
BST INSERT

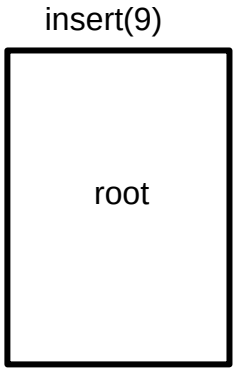
- **The insert function above has a node* &ptr parameter**
  - Since the user of the BST class should never have access to the root pointer, we call the private insert method from the public insert method as follows

```
bool BST::insert(int value)
{   // call private method
    return insert(root, value);
}
```

BST INSERT

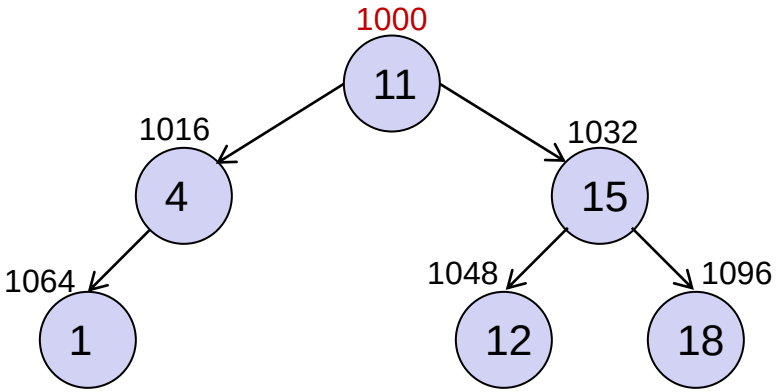
- **Assume we have a BST with nodes at these memory locations**
- **Lets do the box method trace of the call to tree.insert(9)**

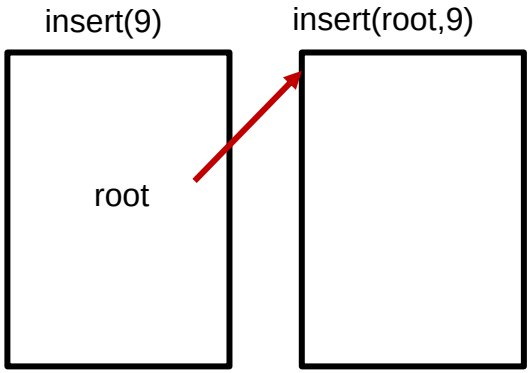




BST INSERT

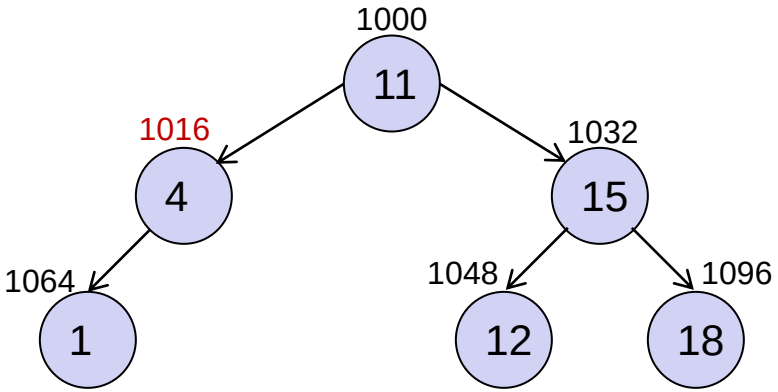
- **The root of the BST is at memory location 1000 so we call tree.insert(root, 9)**

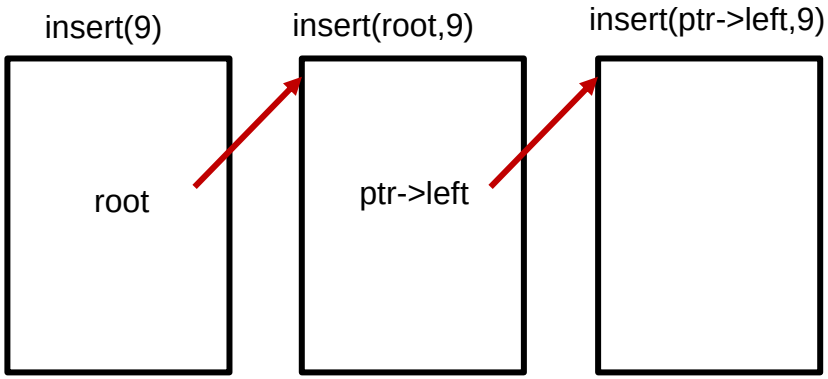




BST INSERT

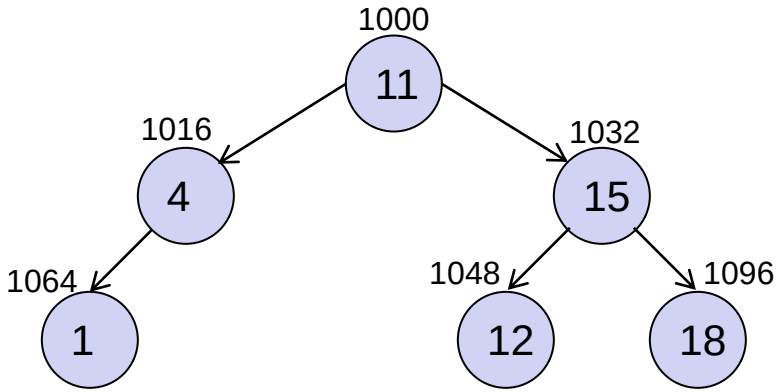
- **ptr->value is 11 > 9 so we go left and call call tree.insert(ptr->left, 9)**

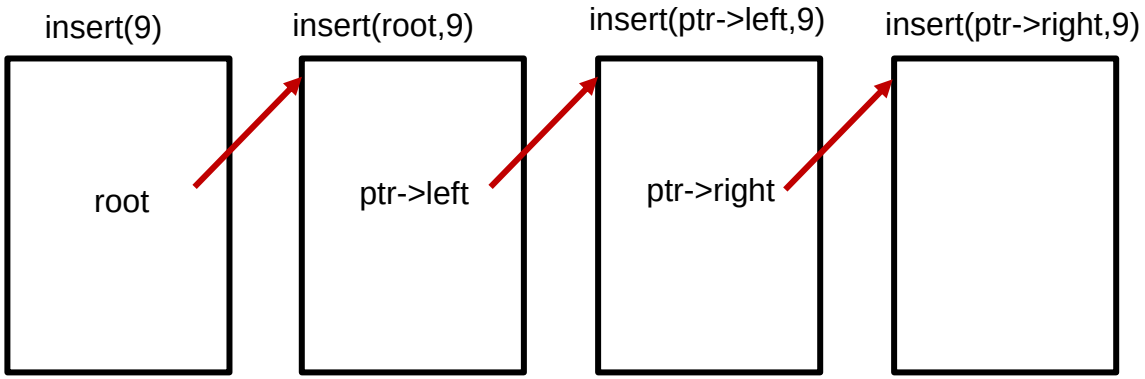




BST INSERT

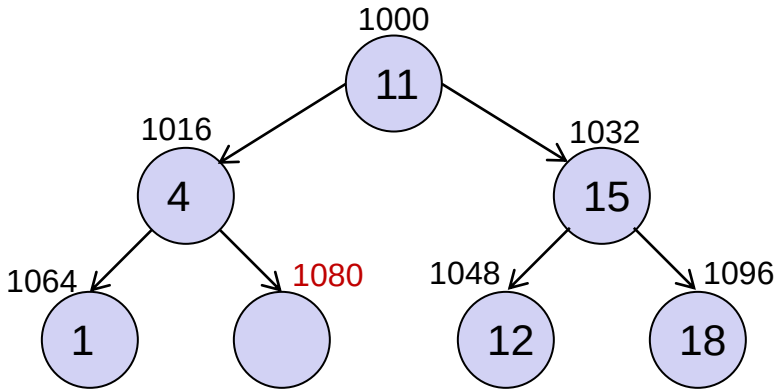
- **ptr->value is 4 < 9 so we go right and call call tree.insert(ptr->right, 9)**

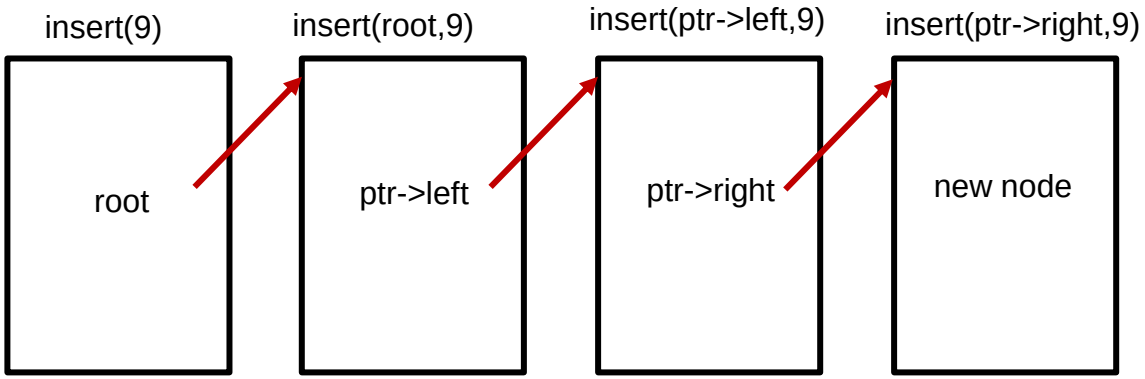




BST INSERT

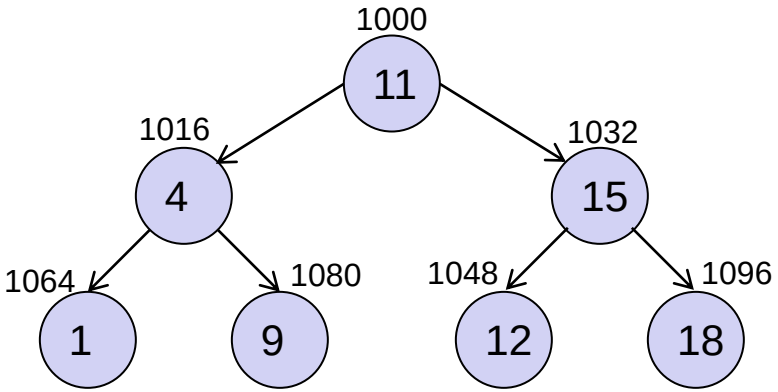
- **Since ptr->right == NULL we create a new tree node at location 1080 and store this in ptr->right which is a reference parameter**

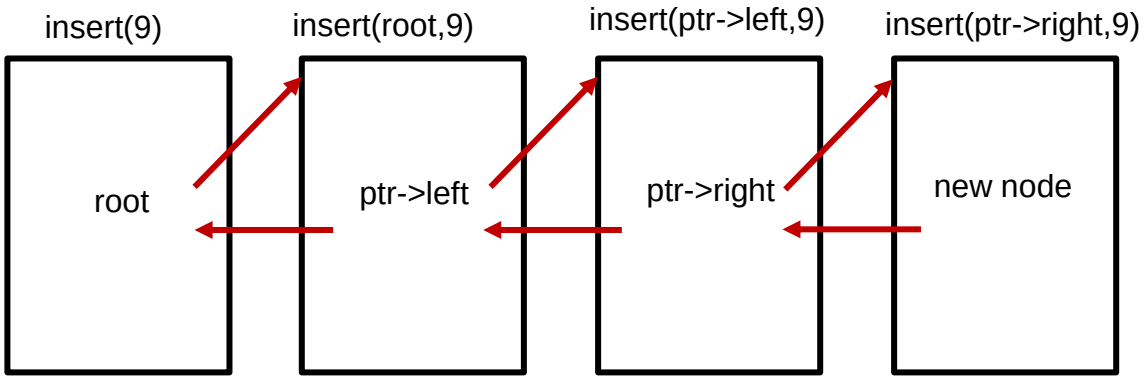




BST INSERT

- **Finally we save value 9 in the new node and return true back to the insert(9) call**





BST INSERT

- If we have a balanced tree, each BST insertion operation will take $O(\log_2 N)$ steps
- If we have a very unbalanced tree, each BST insertion operation may take O(N) steps
- On average, we can expect $O(\log_2 N)$ steps per insertion

BINARY TREES

BST DELETE

BST DELETE

- **Assume we are given a valid BST and we wish to delete a node with a given value**
  - We have to be careful to maintain a valid BST

- **Delete Algorithm:**
  - Start at root of tree
  - Search for node to delete from tree
  - Adjust tree pointers to “jump over” the deleted node
  - Delete the node

BST DELETE

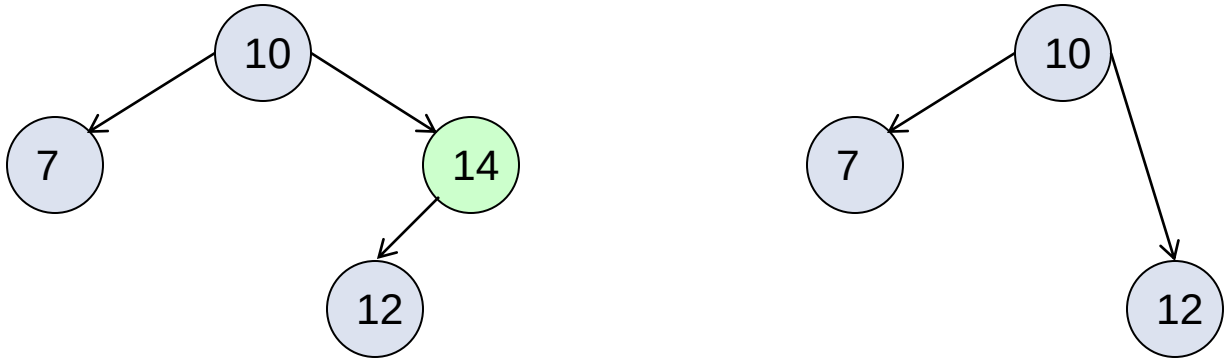
There are three cases to consider when deleting a node:

0 children – set pointer to deleted node to null

delete value 12

BST DELETE

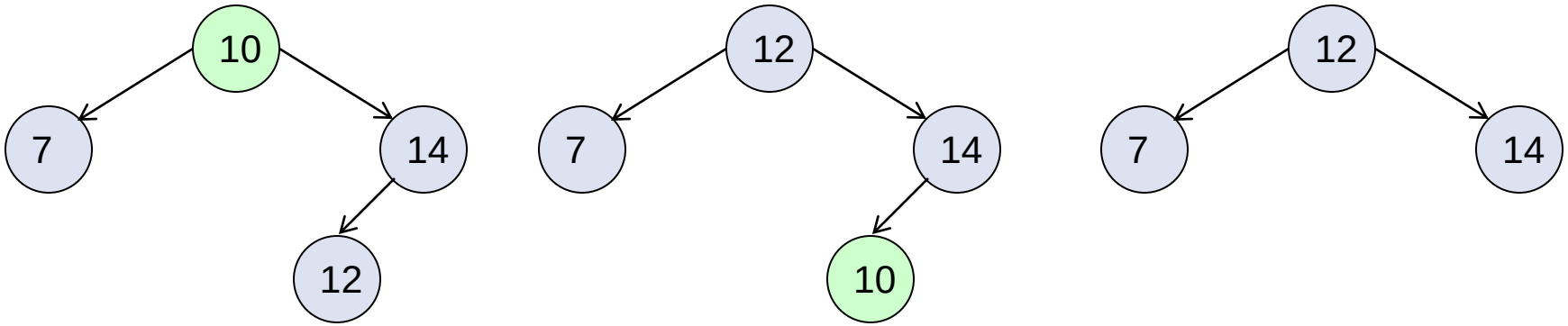
1 child – change pointer in the parent of the deleted node so it points to the child of the deleted node



delete value 14

BST DELETE

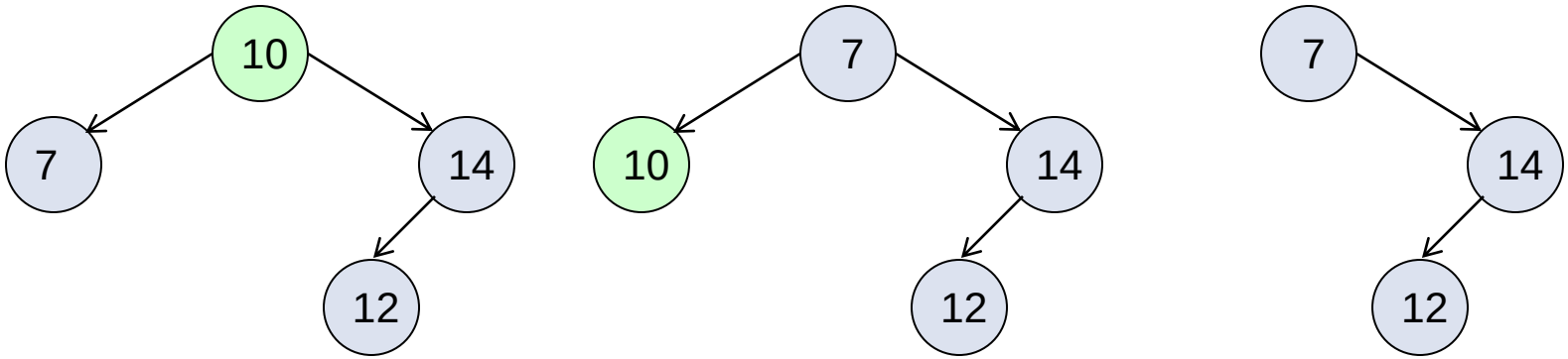
2 children – find left most node in right sub tree
– swap value with node to be deleted
– delete left most node from tree



delete value 10

BST DELETE

2 children – find right most node in left sub tree
– swap value with node to be deleted
– delete right most node from tree



delete value 10

BST DELETE

- **In this case, swapping with left most node in right subtree resulted in a "better looking" BST**

- **If you want to be very fancy, you could check which option yields the most balanced BST before you delete the node**
  - This is fairly complicated to compute
  - Most implementations just pick option A or option B

- **There are better methods for creating balanced BSTs**
  - AVL trees, red-black trees, 2-3 trees, etc.
  - These are beyond the scope of this class

BST DELETE

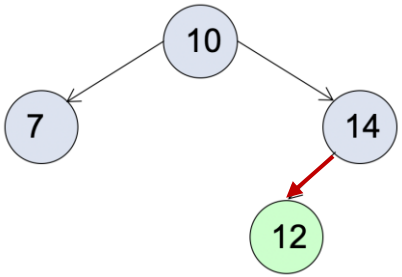
```
bool BST::delete(node* &ptr, int value)
{
    // value not found, so stop
    if (ptr == NULL)
        return false;

    // value found, so delete
    else if (ptr->value == value)
        return delete_node(ptr);    // handle 3 cases
    ...
```

BST DELETE

```
    ...

    // recursive search left
    else if (ptr->value > value)
        return delete(ptr->left, value);

    // recursive search right
    else if (ptr->value < value)
        return delete(ptr->right, value);
}
```

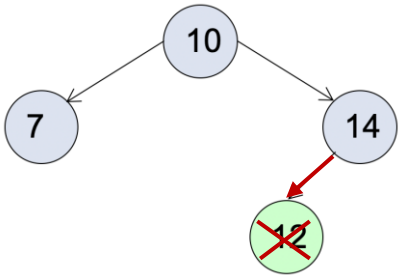
BST DELETE

```
bool BST::delete_node(node * & ptr)
{
    // zero children case
    if ((ptr->left == NULL) && (ptr->right == NULL))
    {
        delete ptr;
        ptr = NULL;
        return true;
    }
    ...
```



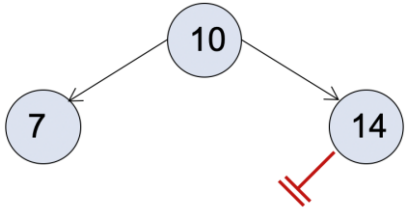
BST DELETE

```
bool BST::delete_node(node * & ptr)
{
    // zero children case
    if ((ptr->left == NULL) && (ptr->right == NULL))
    {
        delete ptr;
        ptr = NULL;
        return true;
    }
    ...
```



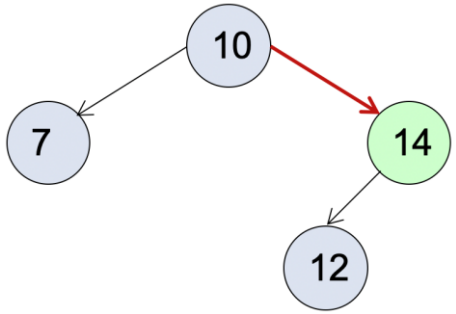
BST DELETE

```
bool BST::delete_node(node * & ptr)
{
    // zero children case
    if ((ptr->left == NULL) && (ptr->right == NULL))
    {
        delete ptr;
        ptr = NULL;
        return true;
    }
    ...
```



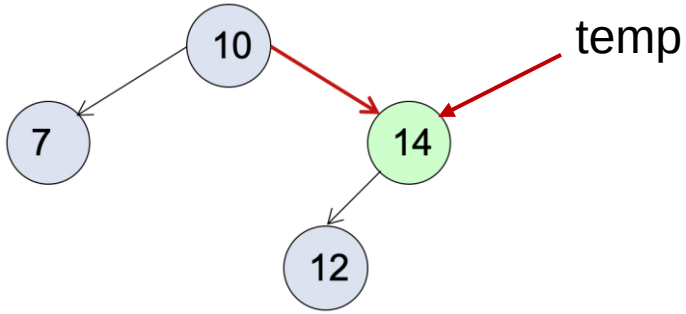
BST DELETE

```
// one child on left
if ((ptr->left != NULL) && (ptr->right == NULL))
{
    node * temp = ptr;
    ptr = ptr->left;
    delete temp;
    return true;
}
...
```



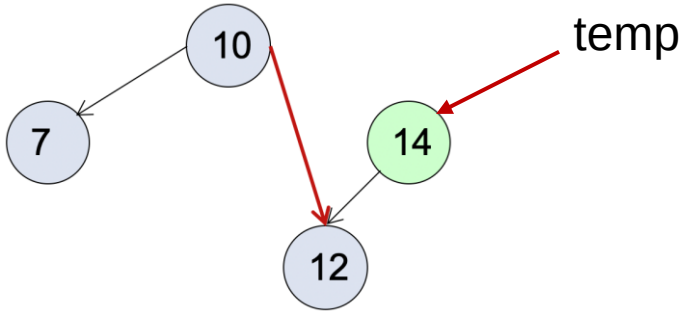
BST DELETE

```
// one child on left
if ((ptr->left != NULL) && (ptr->right == NULL))
{
    node * temp = ptr;
    ptr = ptr->left;
    delete temp;
    return true;
}
...
```



BST DELETE

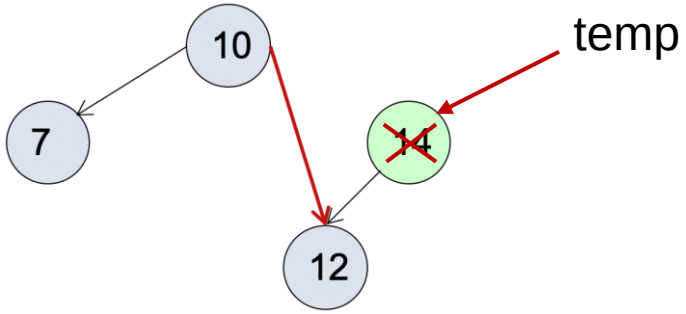
```
// one child on left
if ((ptr->left != NULL) && (ptr->right == NULL))
{
    node * temp = ptr;
    ptr = ptr->left;
    delete temp;
    return true;
}
...
```



This “jumps over”
the deleted node

BST DELETE

```
// one child on left
if ((ptr->left != NULL) && (ptr->right == NULL))
{
    node * temp = ptr;
    ptr = ptr->left;
    delete temp;
    return true
}
...
```

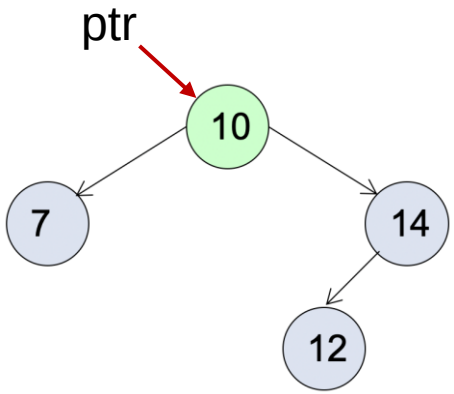


BST DELETE

```
// one child on right
if ((ptr->left == NULL) && (ptr->right != NULL))
{
   node * temp = ptr;
   ptr = ptr->right;
   delete temp;
   return true;
}
...
```

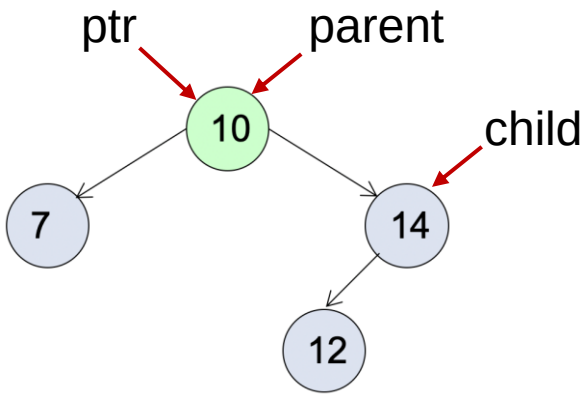
BST DELETE

```
// handle two children
if ((ptr->left != NULL) && (ptr->right != NULL))
{
   // find left most node in right sub tree
   node * parent = ptr;
   node * child = parent->right;
   while (child->left != NULL)
   {
      parent = child;
      child = child->left;
   }
   ...
```



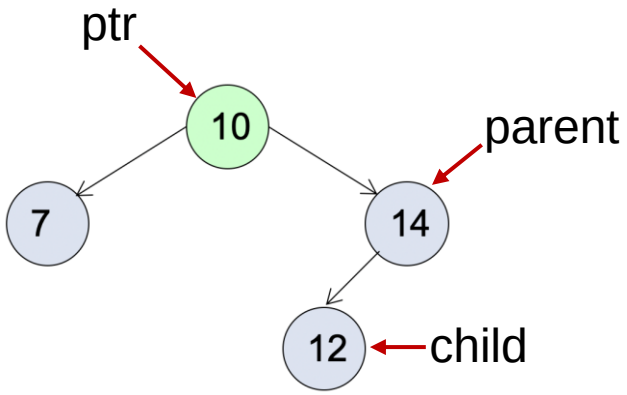
BST DELETE

```
// handle two children
if ((ptr->left != NULL) && (ptr->right != NULL))
{
   // find left most node in right sub tree
   node * parent = ptr;
   node * child = parent->right;
   while (child->left != NULL)
   {
      parent = child;
      child = child->left;
   }
   ...
```



BST DELETE

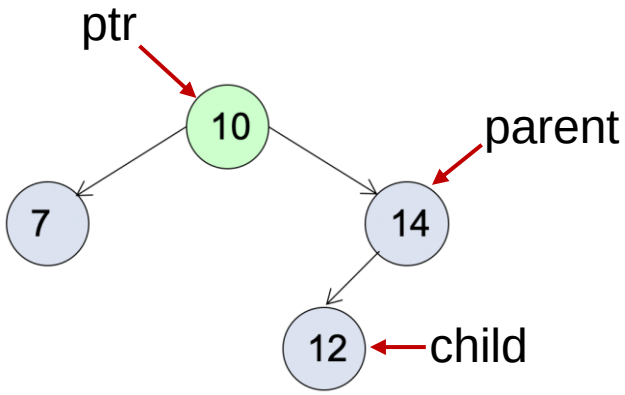
```
// handle two children
if ((ptr->left != NULL) && (ptr->right != NULL))
{
    // find left most node in right sub tree
    node * parent = ptr;
    node * child = parent->right;
    while (child->left != NULL)
    {
        parent = child;
        child = child->left;
    }
    ...
```



This loop stops after only one iteration

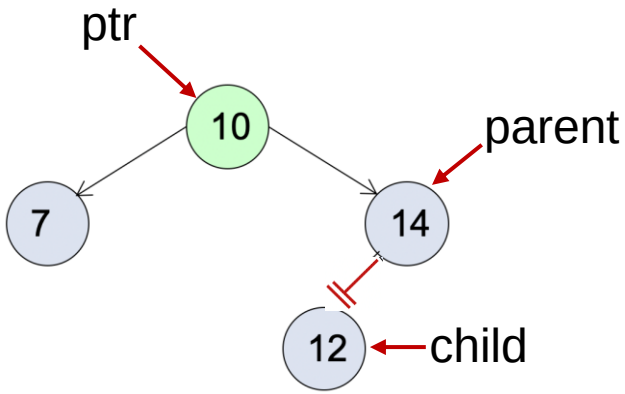
BST DELETE

```
        ...
        // fix pointer to left most node
        if (parent != ptr)
            parent->left = child->right;
        else
            ptr->right = child->right;

        // delete node
        ptr->value = child->value;
        delete child;
        return true;
    }
}
```



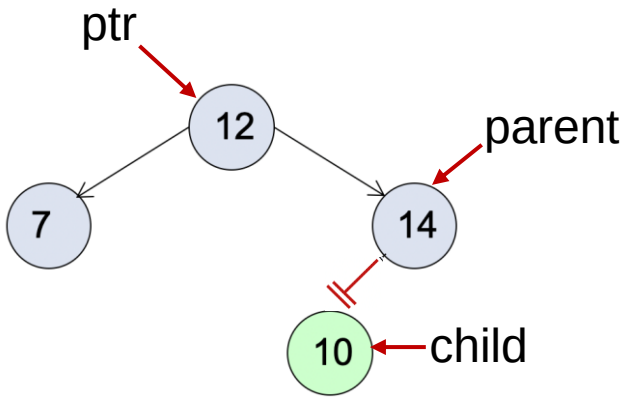
BST DELETE

```
        ...
        // fix pointer to left most node
        if (parent != ptr)
            parent->left = child->right;
        else
            ptr->right = child->right;

        // delete node
        ptr->value = child->value;
        delete child;
        return true;
    }
}
```



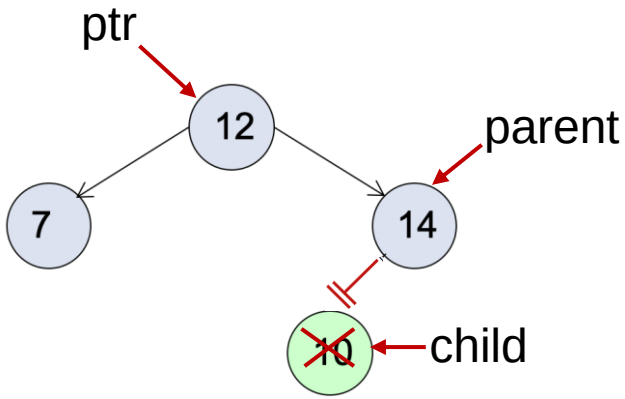
BST DELETE

```
        ...
        // fix pointer to left most node
        if (parent != ptr)
           parent->left = child->right;
        else
           ptr->right = child->right;

        // delete node
        ptr->value = child->value;
        delete child;
        return true;
    }
}
```



BST DELETE

```
        ...
        // fix pointer to left most node
        if (parent != ptr)
           parent->left = child->right;
        else
           ptr->right = child->right;

        // delete node
        ptr->value = child->value;
        delete child;
        return true;
    }
}
```



BST DELETE

```
        ...
        // fix pointer to left most node
        if (parent != ptr)
           parent->left = child->right;
        else
           ptr->right = child->right;

        // delete node
        ptr->value = child->value;
        delete child;
        return true;
    }
}
```

This code handles the special case where the node to right of ptr is the leftmost node

BST DELETE

- **The delete function above has a node* &ptr parameter**
  - Since the user of the BST class should never have access to the root pointer, we call the private delete method from the public delete method as follows

```
bool BST::delete(int value)
{   // call private method
    return delete(root, value);
}
```

BST DELETE

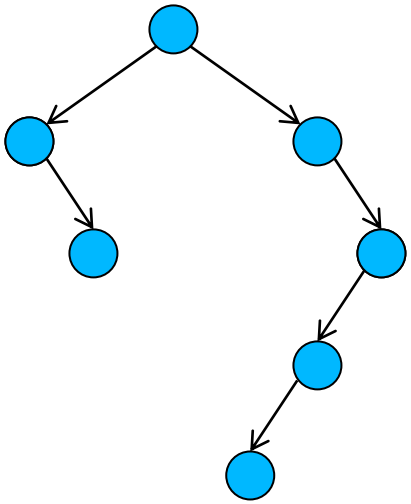
- If we have a balanced tree, each BST delete operation will take $O(\log_2 N)$ steps
- If we have a very unbalanced tree, each BST delete operation may take O(N) steps
- On average, we can expect $O(\log_2 N)$ steps per deletion

BINARY TREES

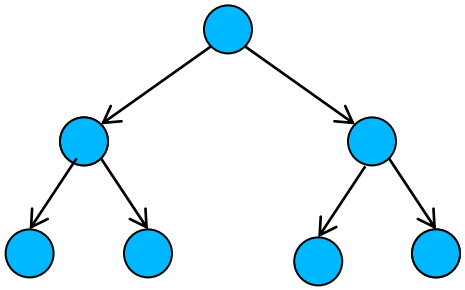
BST BALANCE

BST BALANCE

- **After many insert/delete operations we may end up with a very unbalanced BST, which will slow down search**
  - How can we balance the BST again?



Not balanced

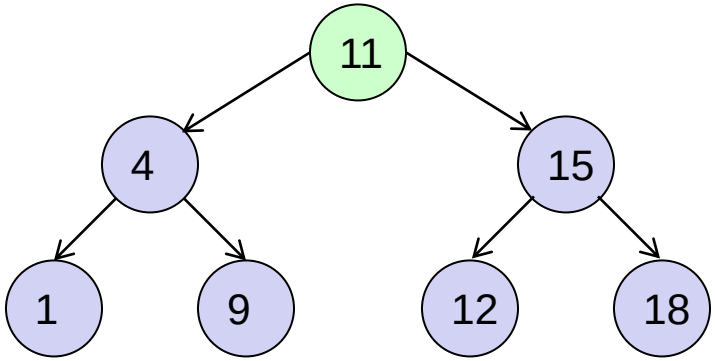


Balanced

BST BALANCE

- **The trick is to look at how we access the array when we perform binary search on sorted data**

1	4	9	11	12	15	18
0	1	2	3	4	5	6

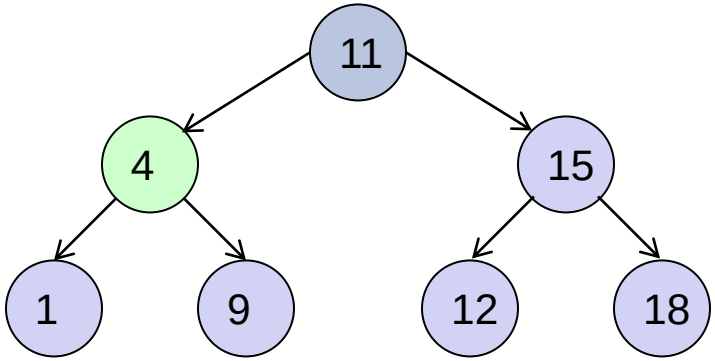


- When we search this array we always visit data[3]=11 first
- We want to insert the value 11 into our balanced BST first

BST BALANCE

- **The trick is to look at how we access the array when we perform binary search on sorted data**

1	4	9	11	12	15	18
0	1	2	3	4	5	6

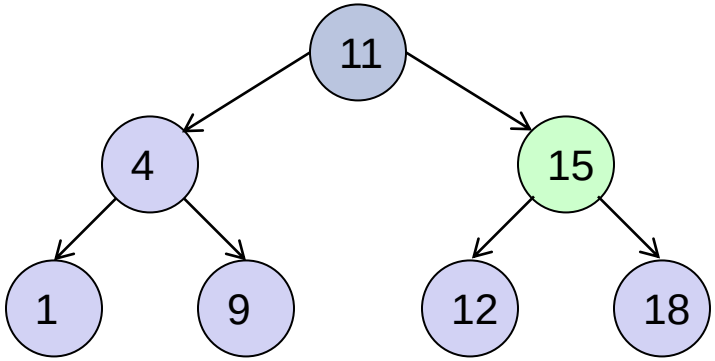


- If value<11 we will visit data[1]=4 next
- We want to insert 4 into our BST before values 1 and 9

BST BALANCE

- **The trick is to look at how we access the array when we perform binary search on sorted data**

1	4	9	11	12	15	18
0	1	2	3	4	5	6

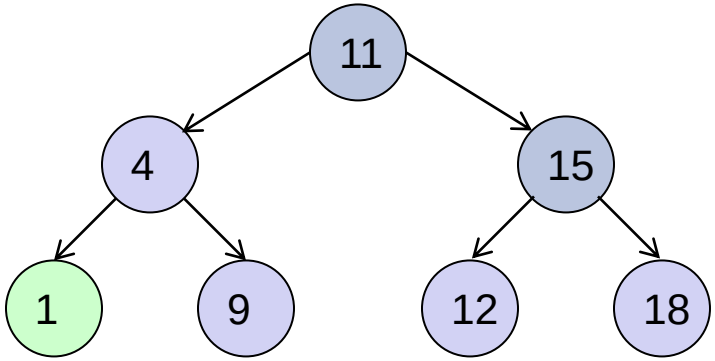


- If value>11 we will visit data[5]=15 next
- We want to insert 15 into our BST before values 12 and 18

BST BALANCE

- **The trick is to look at how we access the array when we perform binary search on sorted data**

1	4	9	11	12	15	18
0	1	2	3	4	5	6



- If value<4 we will visit data[0]=1 next
- We want to insert 1 into our BST after inserting 11 and 4
- The other leaf nodes 9, 12, 18 should also be inserted after their parents

BST BALANCE

- **The first step in the balance process is to extract all of the tree data and store it in sorted order in an array**

- **Extract Algorithm (very similar to sorted print)**
  - Create array large enough for all the data
  - Initialize array index = 0
  - Initialize tree ptr = root
  - Recursively extract data from ptr->left subtree into array
  - Store data at ptr->value in array location data[index]
  - Increment the array index by one
  - Recursively extract data from ptr->right subtree into array

BST BALANCE

```
void BST::extract(node* ptr,
    int data[], int &index)
{
    // terminating condition
    if (ptr == NULL) return;

    // recursive calls to left and right
    extract(ptr->left, data, index);
    data[index++] = ptr->value;
    extract(ptr->right, data, index);
}
```

Index is a reference parameter that is incremented every time we store a value in data array

The recursion here is very similar to a sorted order print

BST BALANCE

- **The next step is to insert sorted data into an empty BST in an order that mimics the order we do binary search**

- **Balance Algorithm**
  - Start with an empty BST
  - Find value at location mid=(low+high)/2 of sorted array
  - Insert this data value into the balanced BST
  - Recursively insert data from [low..mid-1] into left subtree
  - Recursively insert data from [mid+1..high] into right subtree
  - Stop recursion when low > high

BST BALANCE

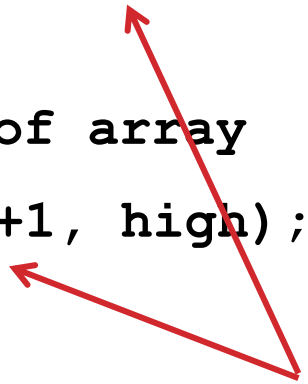
```
void BST::balance(node* &ptr,
    int data[], int low, int high)
{
    // terminating condition
    if (low > high) return;

    // insert middle value
    int mid = (low + high) / 2;
    insert(ptr, data[mid]);
...
```

Check terminating condition

Insert data at mid location first

BST BALANCE

```
...
    // insert data on left half of array
    balance(ptr->left, data, low, mid-1);

    // insert data on right half of array
    balance(ptr->right, data, mid+1, high);
}
```



Use the same array sub-ranges as binary search, but process both halves of array

BST BALANCE

```
void BST::balance_tree()
{
    // extract data in sorted order
    int data[Count];
    int count = 0;
    extract(Root, data, count);
...
```

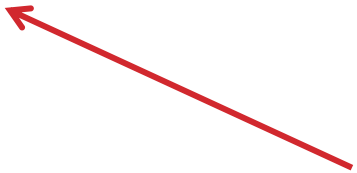
Create an empty array

Recursively extract data in sorted order

BST BALANCE

```
...
    // call recursive function to insert data
    Root = NULL;
    Count = 0;
    balance(Root, data, 0, count-1);
}
```

Create an empty BST



Recursively insert data from the sorted array

BST BALANCE

- **How many operations are needed by this algorithm to balance a binary search tree with N nodes?**
  - We have to traverse whole tree to create sorted array
  - There are N nodes, so this takes O(N) steps
  - We have to insert all N values back into an empty BST
  - Each insert is O(logN) so this takes O(N logN) steps
  - Hence our balance operation is O(N logN)
  - This is very expensive compared to O(logN) search or even O(N) search when the BST is very unbalanced
  - Hence you should only do this when you think the BST is stable and will not be changing much in future

BINARY TREES

SUMMARY

SUMMARY

- **In this section, we introduced the binary tree as an abstract data type and the following tree terminology**

Term	Description
Root node	The root node is at the top of the tree
Leaf node	Leaf nodes are at bottom of the tree and have no children
Internal node	Internal nodes have one or more children nodes
Height of tree	The number of nodes on longest path from root to leaf
Binary tree	A tree with 0,1,2 children per node
Empty tree	A tree with zero nodes
Full tree	A binary tree where root and internal nodes all have 2 children
Complete tree	A binary tree that is full except bottom level filled in from L-R
Balanced tree	A binary tree where height of left and right subtrees differ by <= 1
Binary search tree	A binary tree where all nodes in left subtree are less than parent, and nodes in right subtree are greater than parent

SUMMARY

- **We also described how to define a binary search tree and how to implement the following operations**

Operation	Description	Ave Speed
Print	Start at root of tree, recursively print values in left subtree, print current node, recursively print right subtree	O(N)
Search	Start at root of tree, resursively search left subtree or right subtree depending on value of current node	O(logN)
Insert	Perform recursivse search algorithm, insert new node at the “dead end“ where we were expecting the new value	O(logN)
Delete	Perform recursive search algorithm to find value, adjust pointer to “jump over“ this node according to number of children nodes (0,1,2) and then delete this node	O(logN)
Balance	Extract all data from tree in sorted order, insert data back into an empty tree in “binary search” order (inserting midpoint value before resursively inserting data on left and right halves of the sorted data array)	O(N logN)

SUMMARY

- **Trees are used for many other purposes in computing**
  - We can implement tree sort by inserting N items into a BST and then printing them in O(N logN) time (see demo)
  - Compilers can store arithmetic expressions in a parse tree and then walk this tree to evaluate the expression
  - Binary trees are also used to implement Huffman coding and other data encoding/encryption techniques

- **In the next section, we will introduce heaps, another type of binary tree that is used to store and retrieve data**