Running times?

- printPreOrder( ) O(n)
- printInOrder( ) O(n)
- printPostOrder( ) O(n)
- size( ) O(n)
- height( ) O(n)
- duplicate( ) O(n)

Alternative Design

- Can achieve more flexibility in what is done on a traversal,without the full generality of iterators, by using a functor to specify what is done when a node is visited, e.g.:

```
template <class Object>
template<  class Visitor>
void BinaryTreeNode<Object>::preorder (Visitor visit)
  {
      visit(this); //instead of cout << element
      if (left != NULL) left->preorder(visit)
      if (right!= NULL) right->preorder(visit);
       ...
  }
```

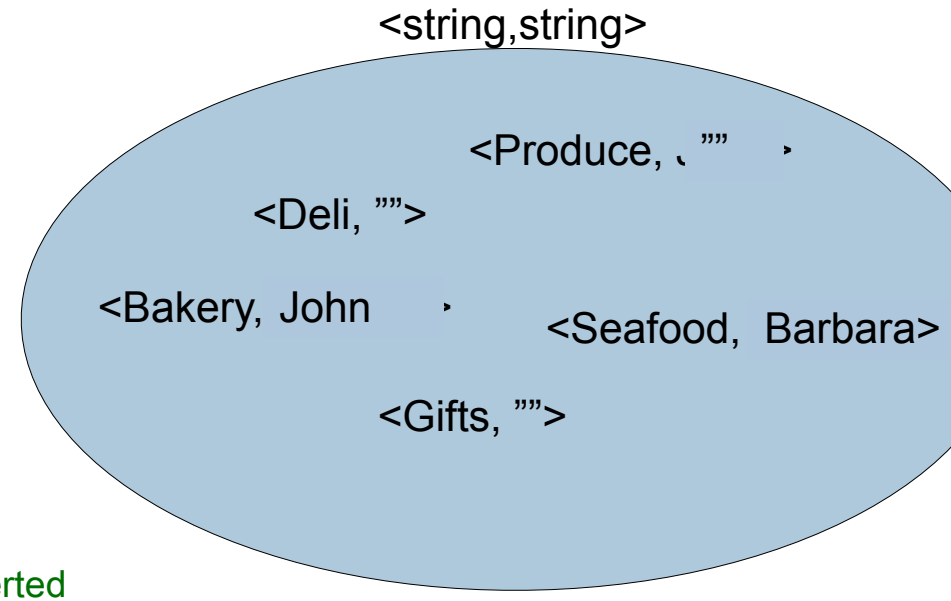
Relationship between height and size

- Some algorithms we'll study have running time O(h). How does this relate to the size n?
- Worst case (biggest height for a given size): Each node has a single child: h=n-1
- Best case: (smallest height for a given size):
  - Each internal node has two children
  - $n = 2^{(h+1)} - 1$,
  - so $h=O(\log n)$
- Average case: $h = O(\log n)$
  - Bigger constant than best case

Lecture 16

Binary Search Trees

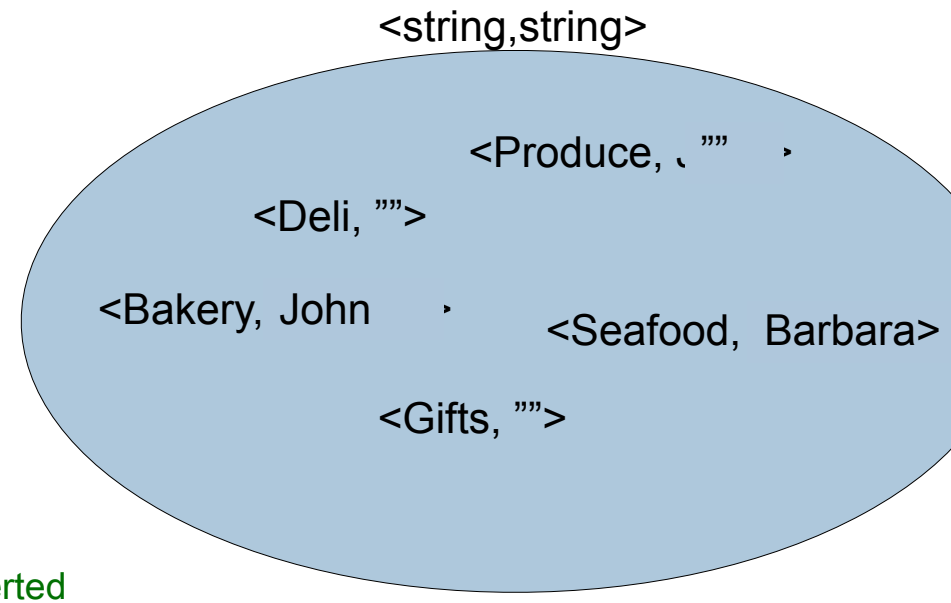
map example




```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

int main ()
{
  map<string,string> mymap;

  mymap["Bakery"]="Barbara";  // new element inserted
  mymap["Seafood"]="Lisa";    // new element inserted
  mymap["Produce"]="John";    // new element inserted

  string name = mymap["Bakery"];   // existing element accessed (read)
  mymap["Seafood"] = name;             // existing element accessed (written)

  mymap["Bakery"] = mymap["Produce"];   // existing elements accessed (read/written)

  name = mymap["Deli"];      // non-existing element: new element "Deli" inserted!

  mymap["Produce"] = mymap["Gifts"];    // new element "Gifts" inserted, "Produce" written
}
```

Modified from http://www.cplusplus.com/reference/unordered_map/unordered_map/operator[]/

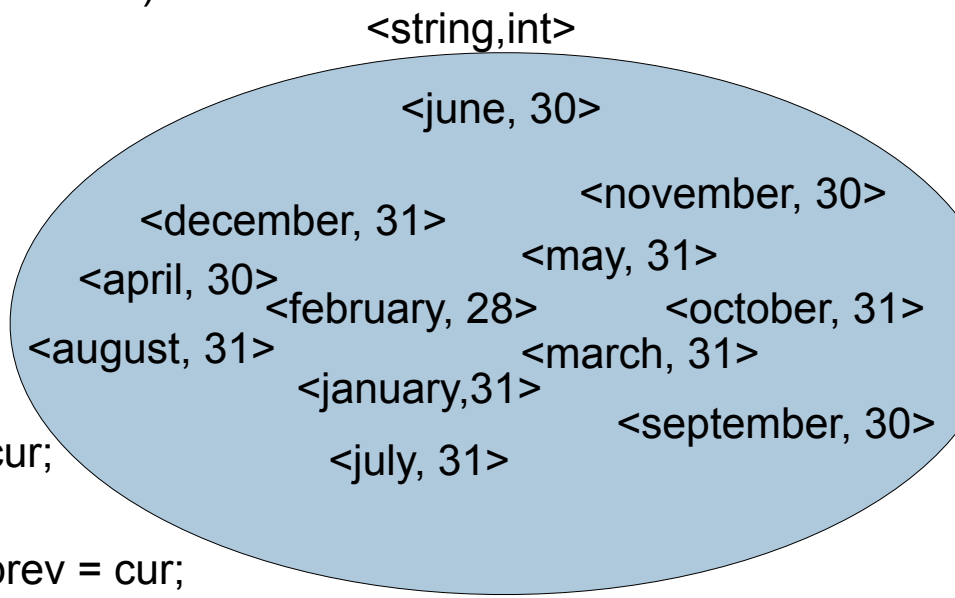
unordered_map example



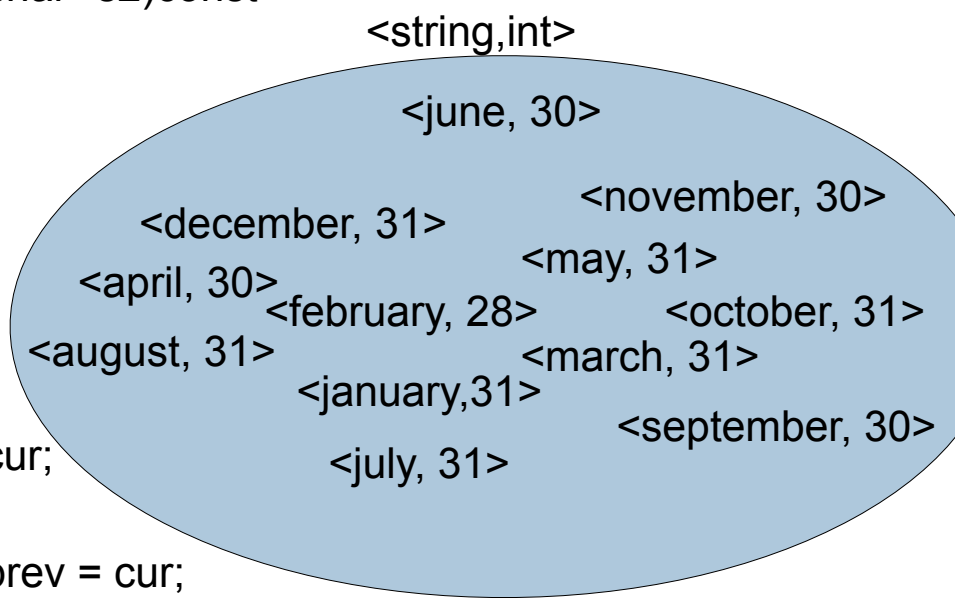

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

int main ()
{
  unordered_map<string,string> mymap;

  mymap["Bakery"]="Barbara";  // new element inserted
  mymap["Seafood"]="Lisa";    // new element inserted
  mymap["Produce"]="John";    // new element inserted

  string name = mymap["Bakery"];   // existing element accessed (read)
  mymap["Seafood"] = name;             // existing element accessed (written)

  mymap["Bakery"] = mymap["Produce"];   // existing elements accessed (read/written)

  name = mymap["Deli"];      // non-existing element: new element "Deli" inserted!

  mymap["Produce"] = mymap["Gifts"];    // new element "Gifts" inserted, "Produce" written
}
```

Modified from http://www.cplusplus.com/reference/unordered_map/unordered_map/operator[]/

```
// Example from SGI STL documentation
struct ltstr{
 bool operator()(const char* s1,const char* s2)const
   {    return strcmp(s1, s2) < 0;  }
 };
int main()  {
  map<const char*, int, ltstr> months;
  months["january"] = 31;
  months["february"] = 28;
  . . .
  map<const char*, int, ltstr>::iterator cur;
  cur  = months.find("june");
  map<const char*, int, ltstr>::iterator prev = cur;
  map<const char*, int, ltstr>::iterator next = cur;
  ++next;
  --prev;
  cout << "Previous (in alphabetical order) is " <<    (*prev).first << endl;
  cout << "Next (in  alphabetical order) is " << (*next).first << endl;
}
```



```
// Example from SGI STL documentation
struct ltstr{
 bool operator()(const char* s1,const char* s2)const
   {    return strcmp(s1, s2) < 0;  }
 };
int main()  {
unordered_ map<const char*, int, ltstr> months;
   months["january"] = 31;
   months["february"] = 28;
   . . .
unordered_ map<const char*, int, ltstr>::iterator cur;
   cur  = months.find("june");
unordered_ map<const char*, int, ltstr>::iterator prev = cur;
unordered_ map<const char*, int, ltstr>::iterator next = cur;
   ++next;
   // cannot go backwards!
   cout << "Previous (in   random      order) is " <<    (*prev).first << endl;


}
```



Binary Search Tree Order Property

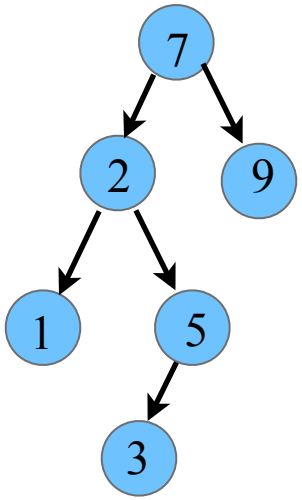
"In a binary search tree, for every node X, all keys in X's left subtree have smaller values than the key in X, and all keys in X's right subtree have larger values than the key in X."

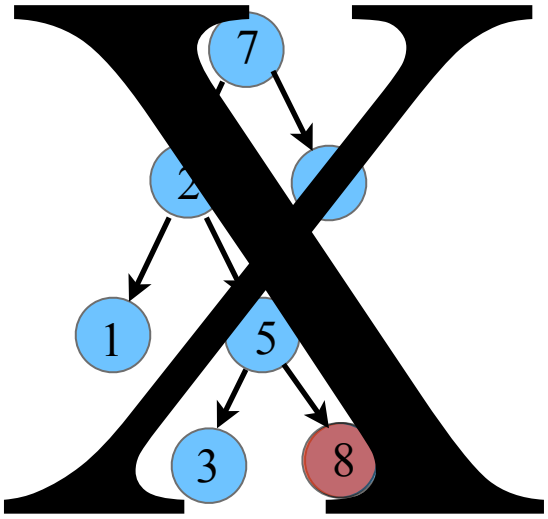
Binary Search Trees

– Binary Trees which store elements in "tree" order
– "Key" of node is element it stores
– Tree order: for each node x in the tree

keys in left subtree < key(x)

keys in right subtree > key(x)



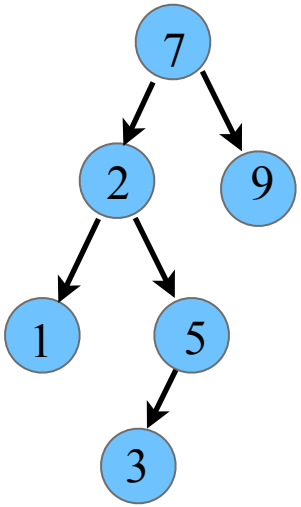


search tree

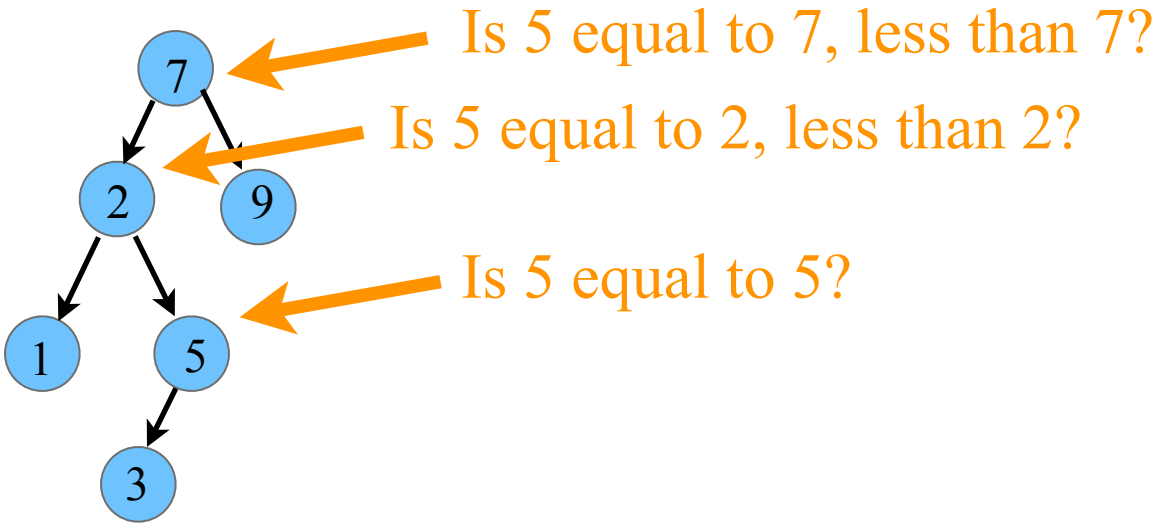
not a search tree

How should we build the binary search tree?

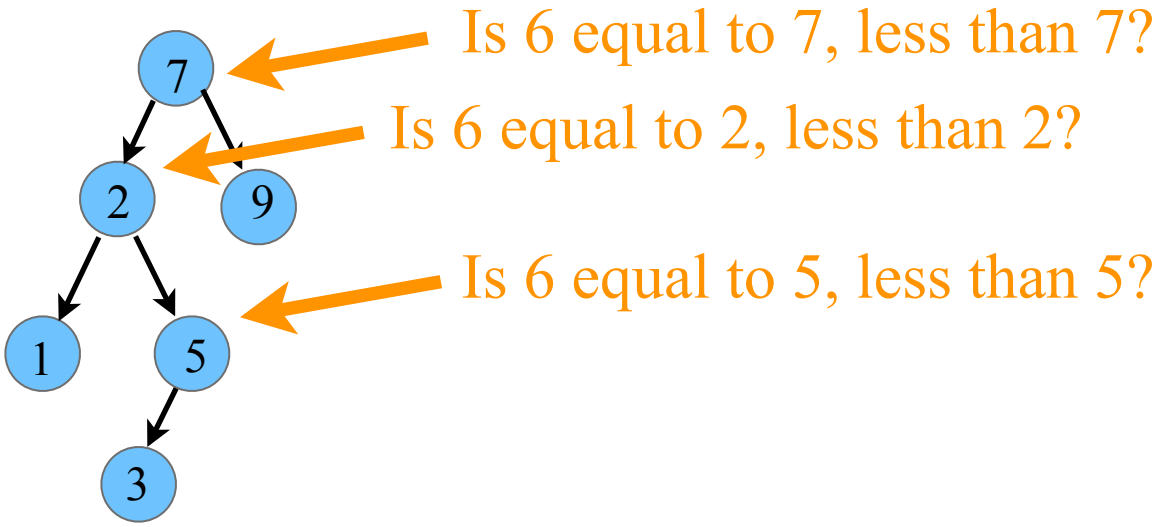
7 2 5 9 1 3



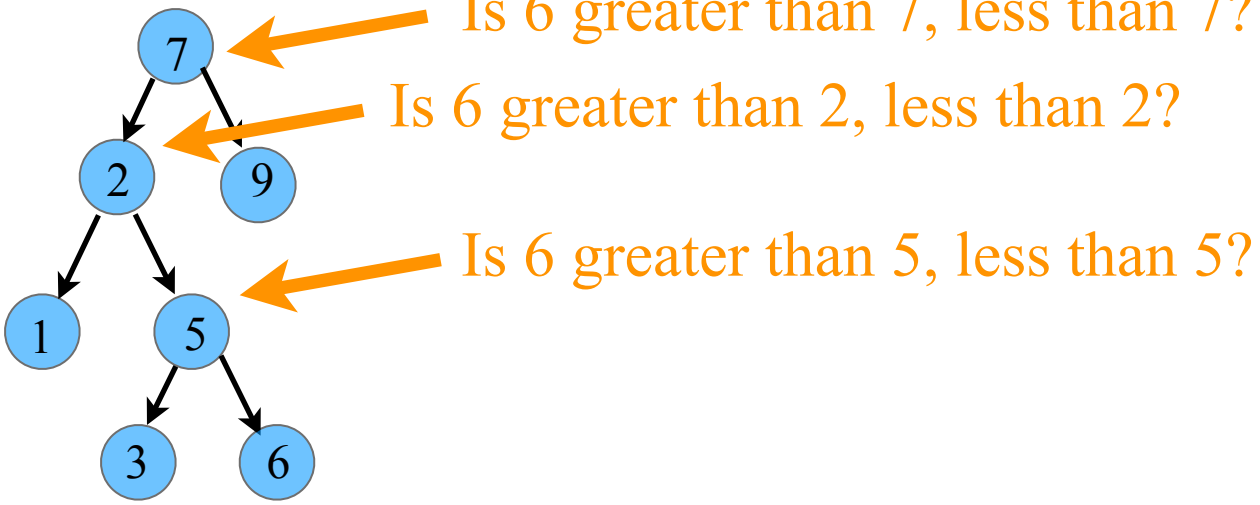
How do we determine if 5 is in the tree?



How do we determine if 6 is in the tree?

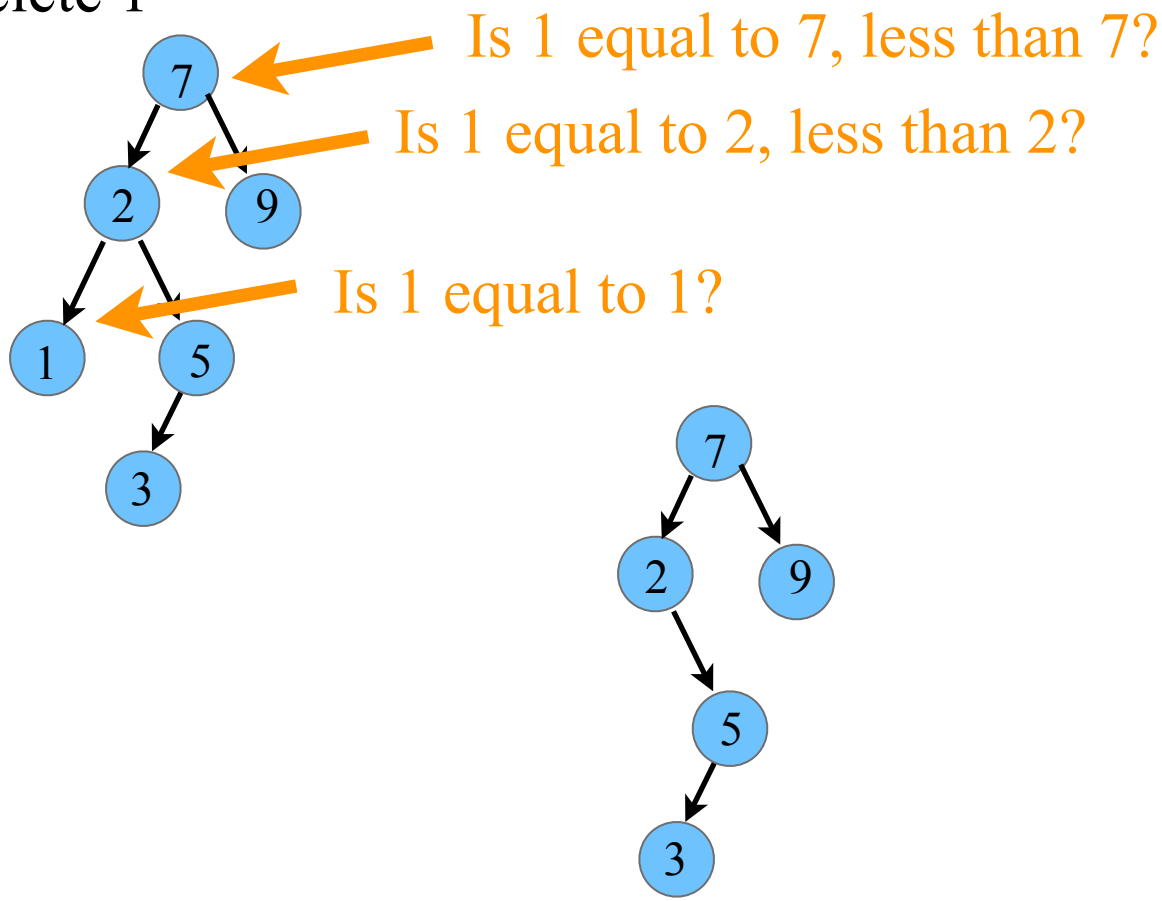


How do we insert 6 into the tree?



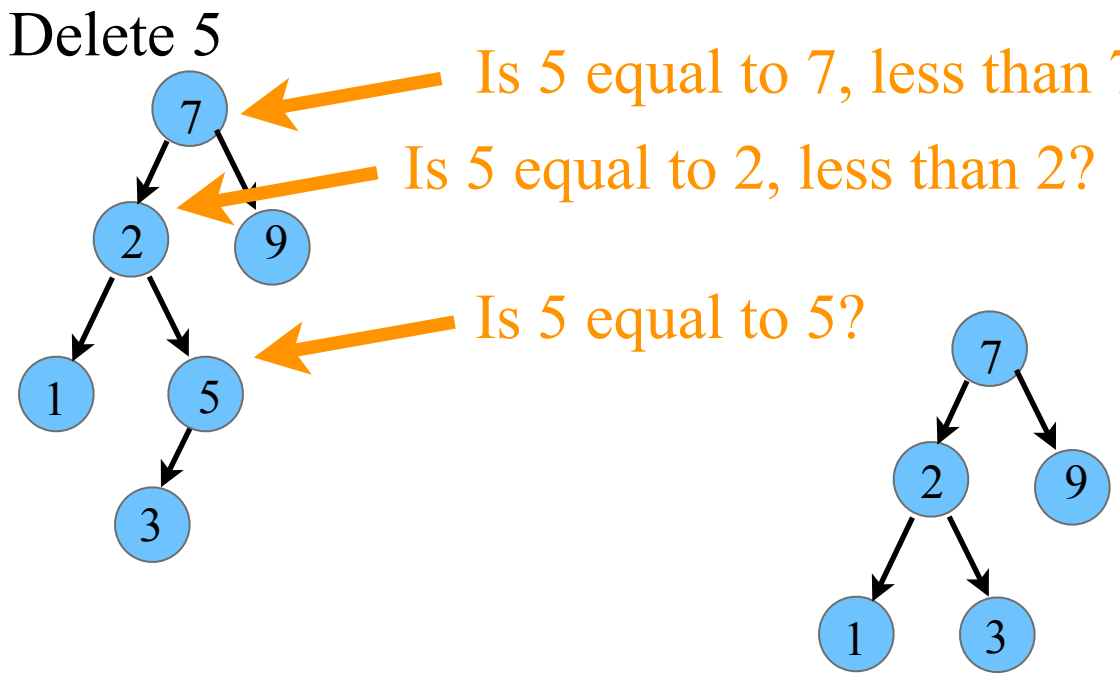
How do we delete an object in the tree

Delete 1



If the node is a leaf, delete the node
and set parent's child pointer to NULL

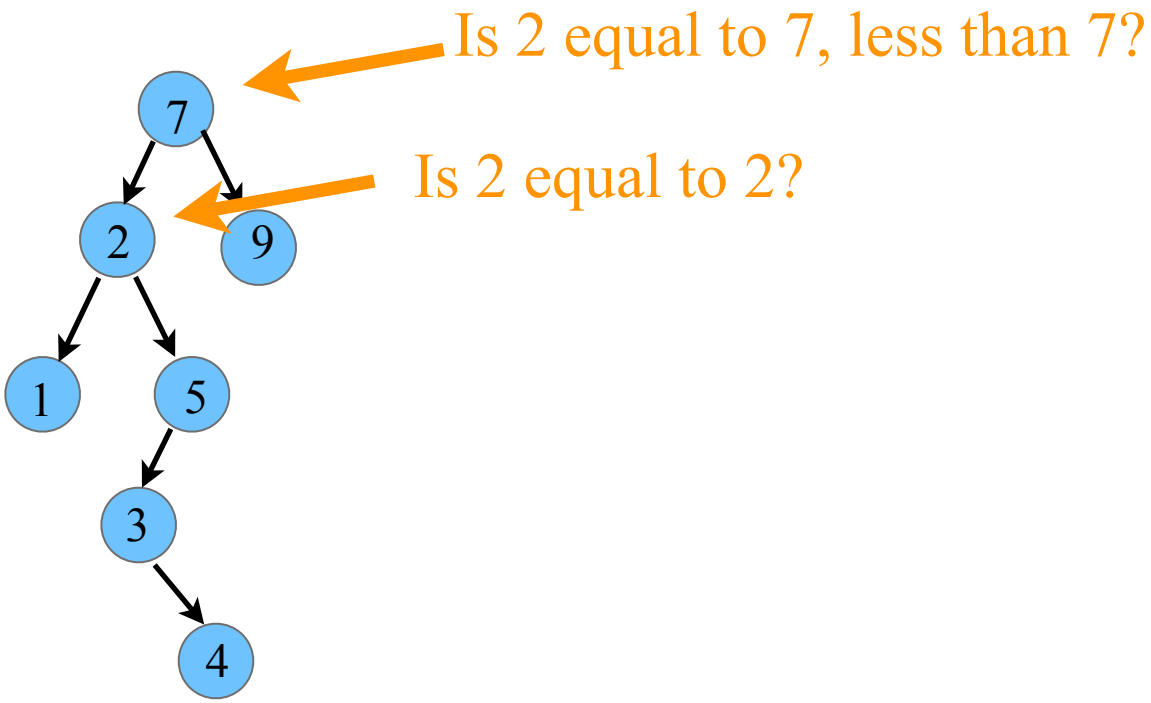
How do we delete an object in the tree that is not a leaf?



If the node has only one child - adjust parent's child link to bypass the node and then delete the node

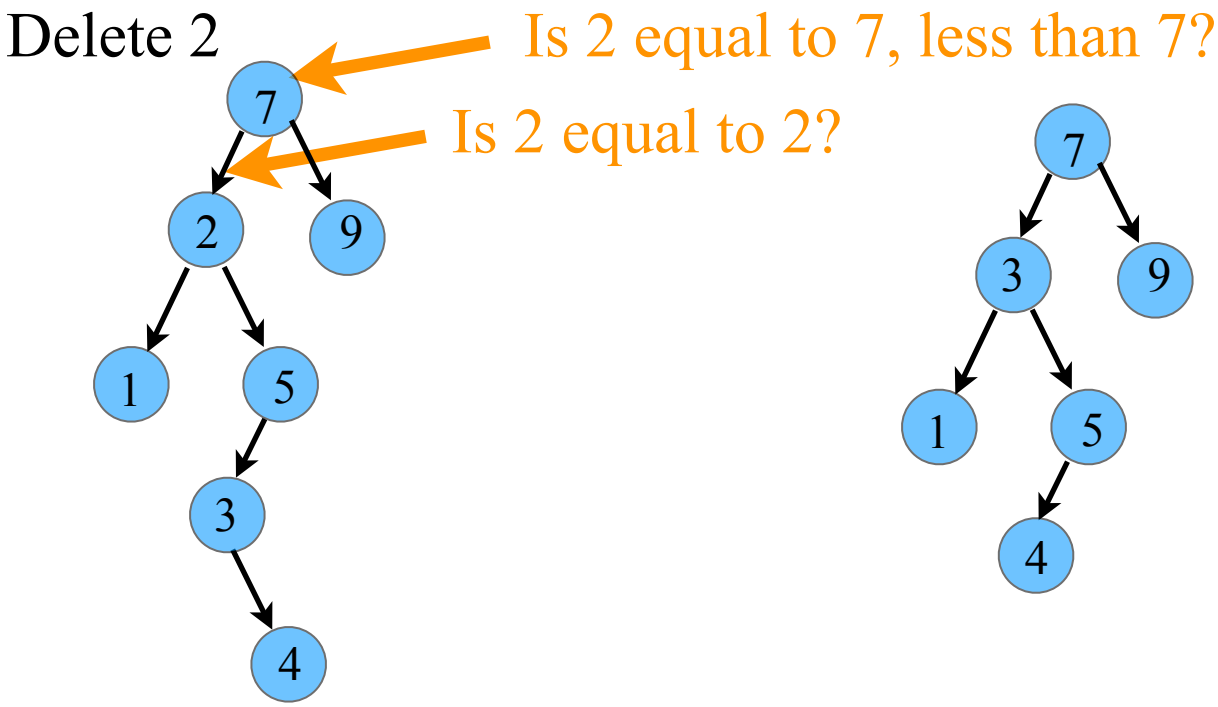
How do we delete an object in the tree that has two children?

Delete 2



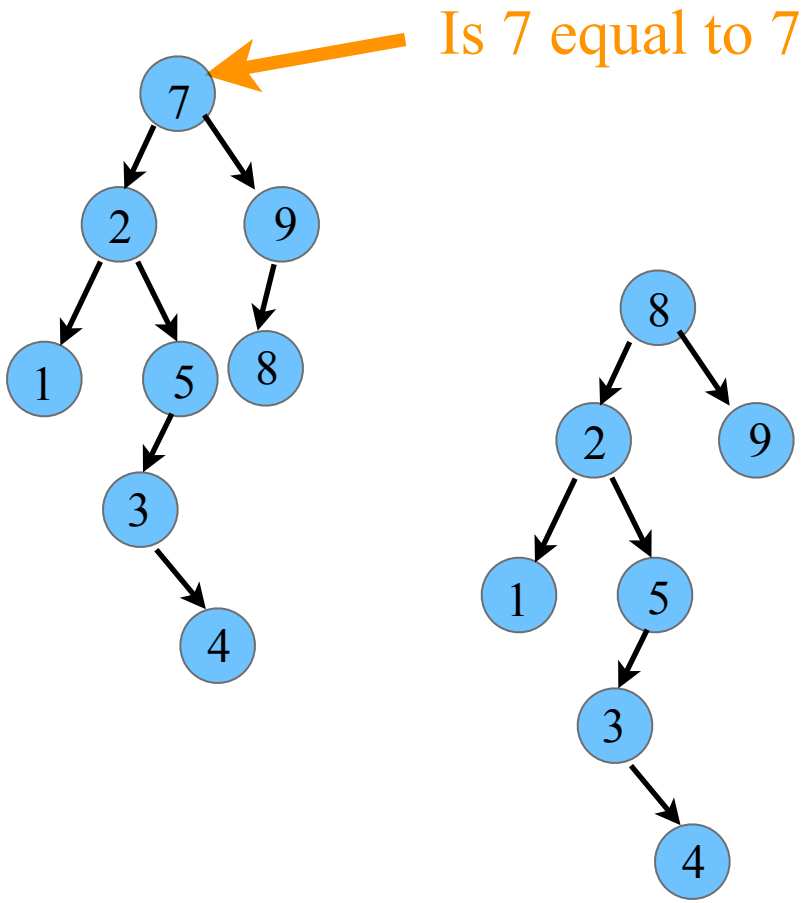
Replace the node with the smallest item in right subtree

How do we delete an object in the tree that has two children?



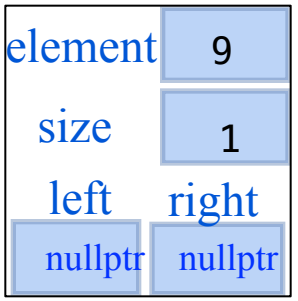
Replace the node with the smallest item in right subtree

Delete 7



Replace the node with the smallest item in right subtree

The Node Class




```
template <class Comparable>
class BinaryNode
{
    Comparable  element;
    BinaryNode *left;
    BinaryNode *right;
    int size;

    BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt, int sz = 1 )
      : element( theElement ), left( lt ), right( rt ), size( sz ) { }
    BinaryNode( Comparable && theElement, BinaryNode *lt, BinaryNode *rt, int sz = 1 )
      : element( std::move(theElement) ), left( lt ), right( rt ), size( sz ) { }

    friend class BinarySearchTree<Comparable>;
};
```

What would a

new BinaryNode<int>(9, nullptr, nullptr, 1);

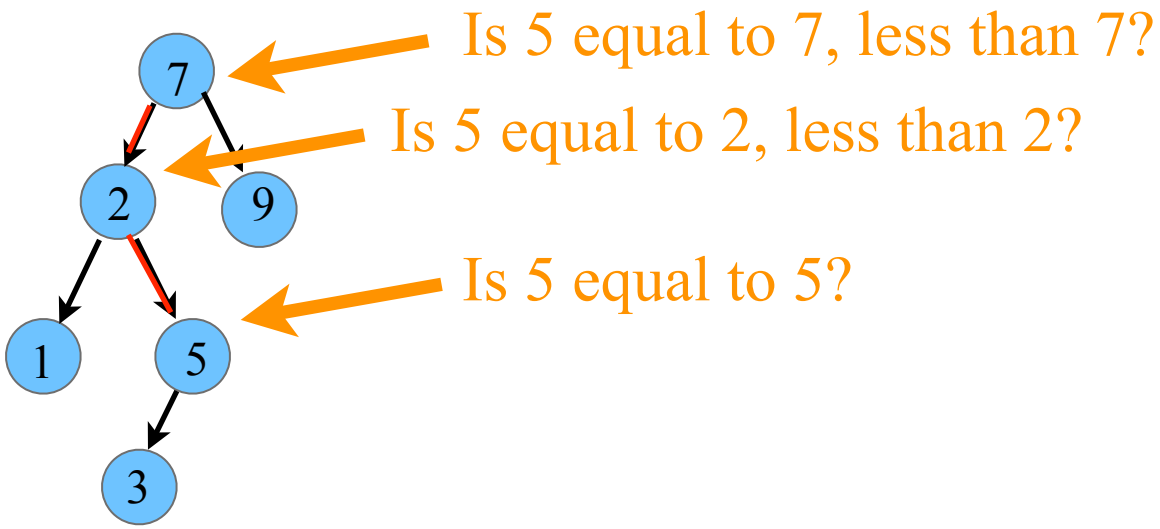
look like if called from the BinarySearchTree class?

The BinarySearchTree Class

```
template <class Comparable>
class BinarySearchTree
{
public:
        typedef BinaryNode<Comparable> Node;

         BinarySearchTree( ) : root( nullptr ) { } // Construct the tree.
          BinarySearchTree( const BinarySearchTree & rhs ) : root( nullptr ) { *this = rhs; }// Copy constructor.
        BinarySearchTree( BinarySearchTree && rhs ); // Move constructor.
        ~BinarySearchTree( ){ makeEmpty( ); } // Destructor for the tree.

         const Comparable & findKth( int k ) const { if( isEmpty( ) ) throw UnderflowException{ }; return findKth( k, root )->element; }
         const Comparable & findMin( ) const { if( isEmpty( ) ) throw UnderflowException{ }; return findMin( root )->element; }
         const Comparable  & findMax( ) const { if( isEmpty( ) ) throw UnderflowException{ }; return findMax( root )->element; }

         bool contains( const Comparable & x ) const { return contains( x, root ); }
         bool isEmpty( ) const { return root == NULL;}
         void makeEmpty( ) { makeEmpty( root ); }
         void insert( const Comparable & x ) { insert( x, root ); }
         void insert( Comparable && x );
         void remove( const Comparable & x ) { remove( x, root ); }
        const BinarySearchTree & operator=( const BinarySearchTree & rhs );

private:
        Node * root;

         int treeSize( Node *t ) const { return t == NULL ? 0 : t->size; }
         void insert( const Comparable & x, Node * & t );
         void remove( const Comparable & x, Node * & t );
         void removeMin( ) { removeMin( root ); }
         void removeMin( Node * & t );
         Node * findMin( Node *t ) const;
         Node * findMax( Node *t ) const;
         Node * find( const Comparable & x, Node *t ) const;
         void makeEmpty( Node * & t );
         Node * clone( Node *t ) const;
         Node *findKth( int k, Node *t ) const;
};
```

Search

How do we determine if 5 is in the tree?



Very similar to binary search in a vector

Searching in BSTs

Search for key x in BST

Start at root node,

repeat the following steps until x is found or node is NULL

Compare x to key at current node

- if x < key, move on to left child
- If key < x, move on to right child
- if equal, done
- Can do recursively
- Faster to do iteratively
  - Fastest methods are careful to limit # of comparisons

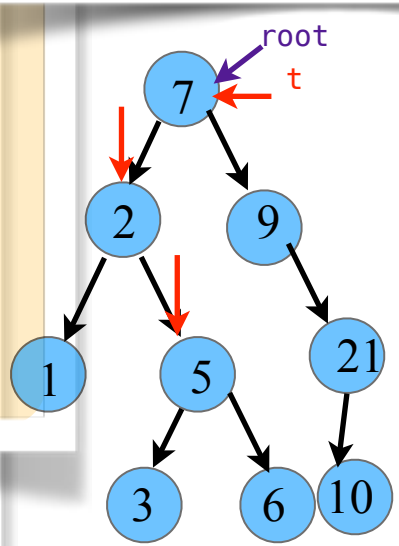
Finding if an object is in the binary search tree

```
template <class Comparable>
class BinarySearchTree
{
public:
    ...// public methods

    bool contains( const Comparable & x ) const
    { return contains( x, root ) ; }

private:
    Node * root;
    ···  // private methods
```

```
    bool contains( const Comparable & x, Node *t ) const
    {
        while( t != nullptr )
            if( x < t->element )
                t = t->left;
            else if( t->element < x )
                t = t->right;
            else
                return true;    // Match

        return false;         // Not found
    }
```

```
};
```

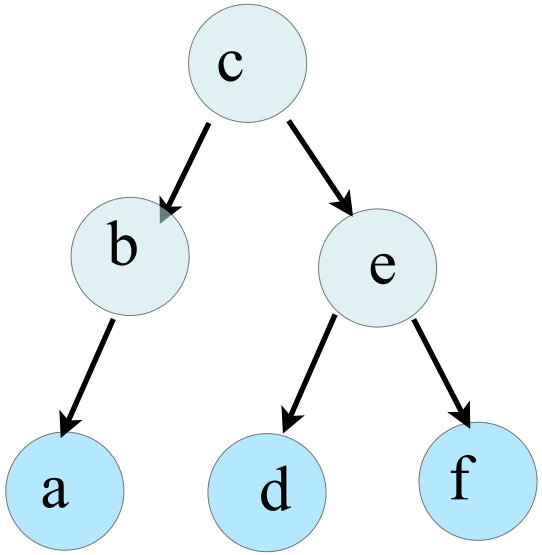
```
Internal method to determine if an item is in
a subtree.
x is item to search for.
t is the node that roots the tree.
Return node containing the matched item.
```

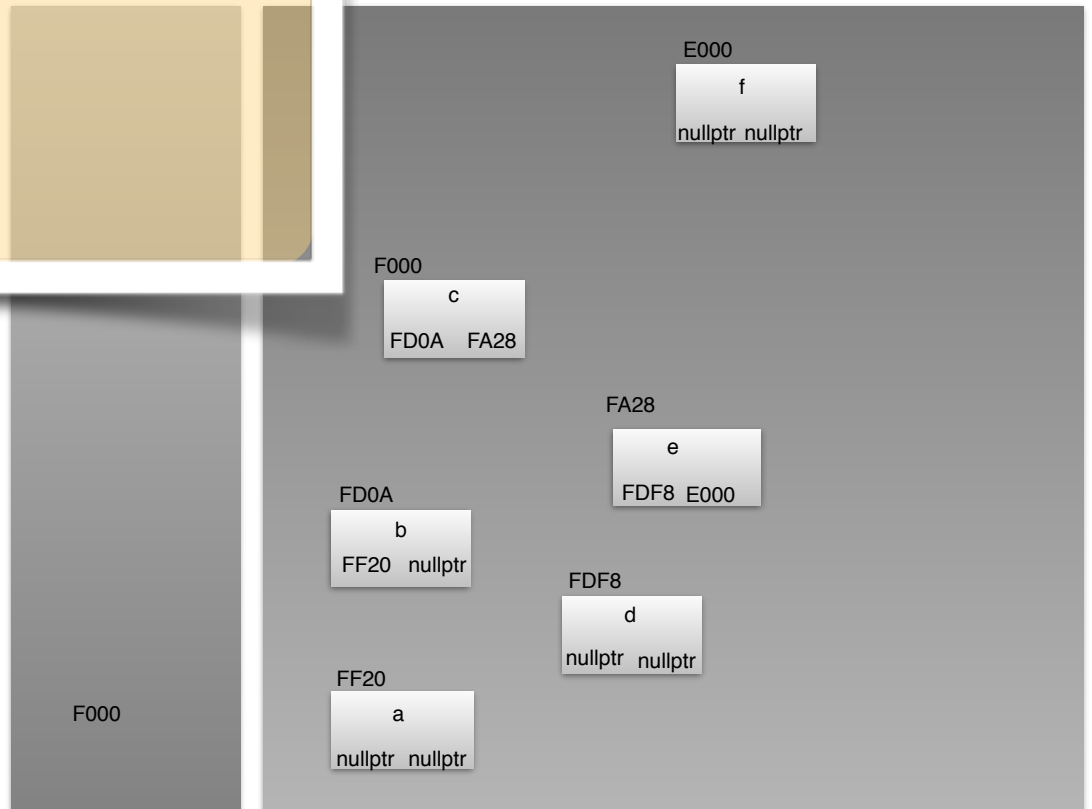


finding 5

```
bool contains( const Comparable & x ) const
{ return contains( x, root ) ; }
```

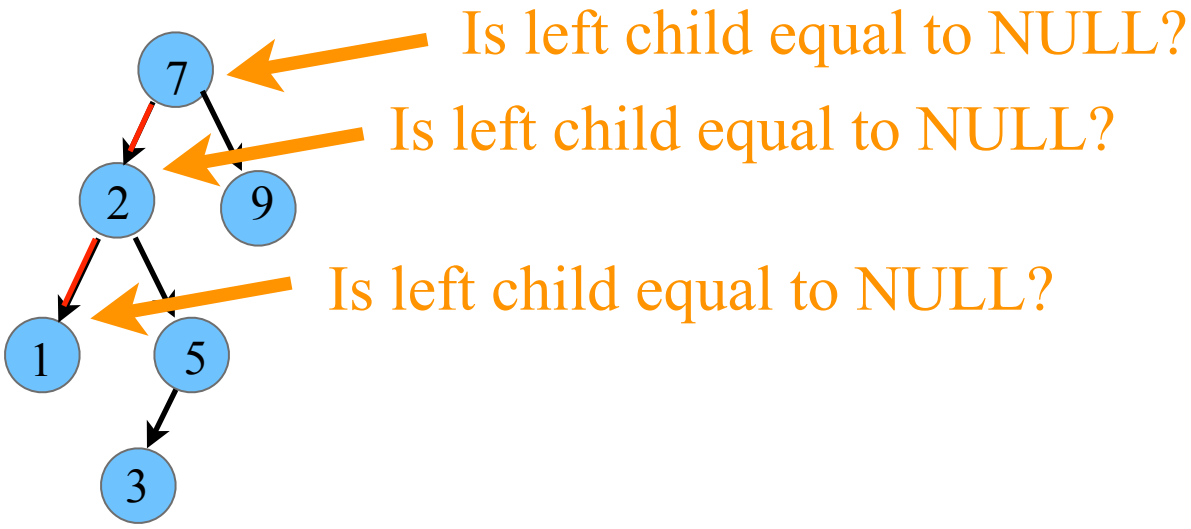
```
bool contains( const Comparable & x, Node *t ) const
{
  while( t != nullptr )
      if( x < t->element )
          t = t->left;
      else if( t->element < x )
          t = t->right;
      else
          return true;    // Match

  return false;          // Not found
}
```





Special Search

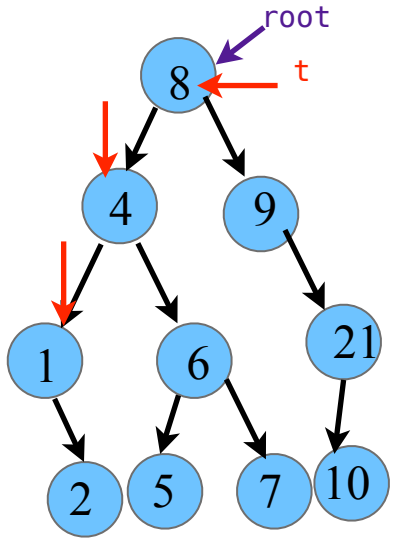
Finding the smallest object in the tree



Finding the minimum object in the binary search tree

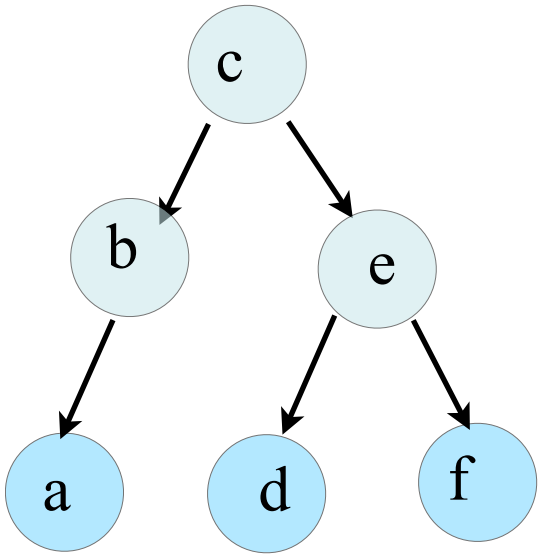
```
template <class Comparable>
class BinarySearchTree
{
public:
     ...// public methods
    const Comparable  & findMin( ) const
    {
      if( isEmpty( ) )
         throw UnderflowException{ };
      return findMin( root )->element;
    }
private:
    Node * root;
    ...  // private methods


  Node * findMin( Node *t ) const
  {

       while( t->left != nullptr )
           t = t->left;

       return t;
  }


};
```

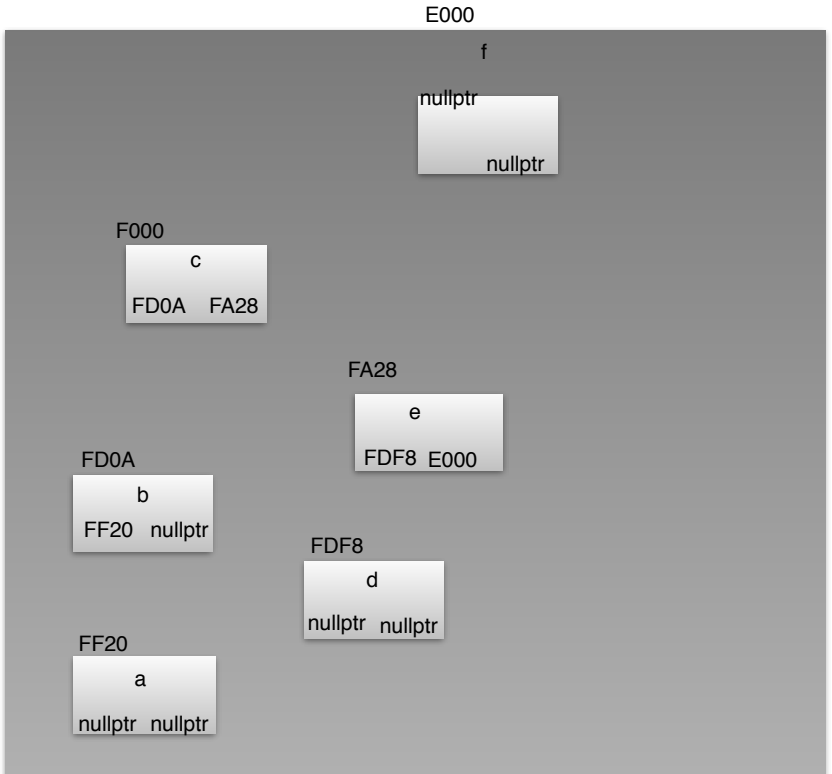
```
Internal method to find the smallest item
in a subtree t.
Return node containing the smallest item.
```



```
const Comparable  & findMin( ) const
{
  if( isEmpty( ) )
     throw UnderflowException{ };
  return findMin( root )->element;
}
```

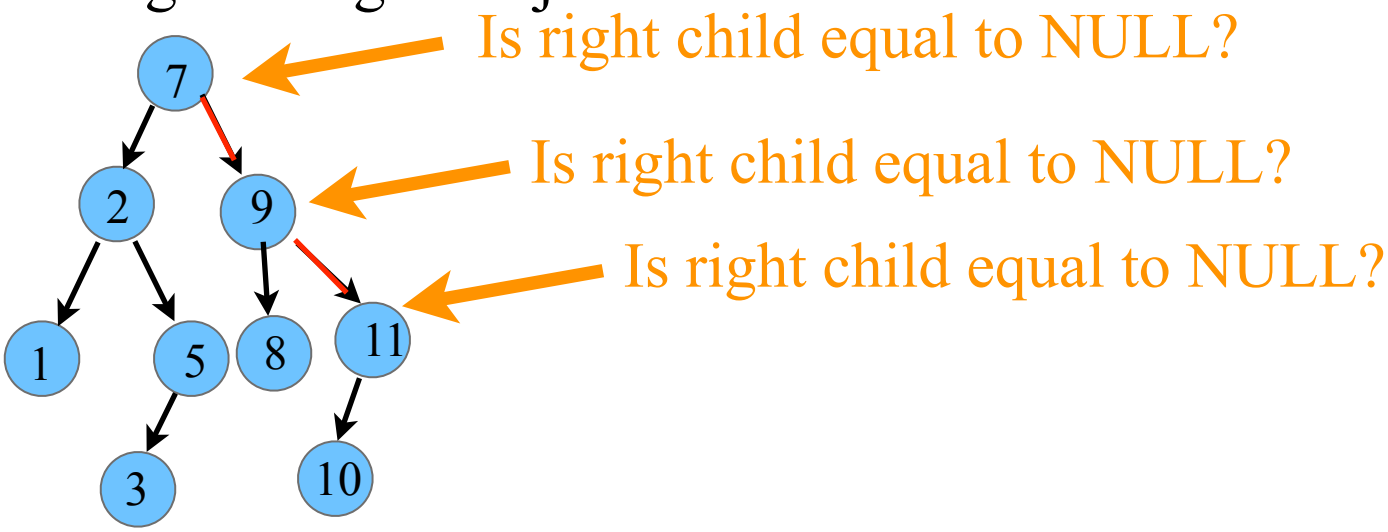
```
Node * findMin( Node *t ) const
{
   while( t->left != nullptr )
       t = t->left;

   return t;
}
```





Special Search

Finding the largest object in the tree

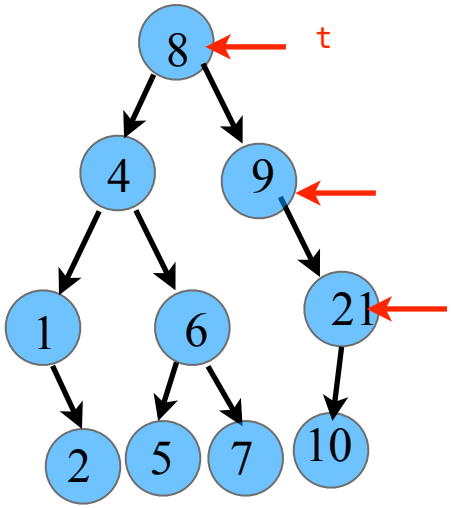


Finding the maximum object in the binary search tree

```
template <class Comparable>
class BinarySearchTree
{
public:
     ...// public methods
   const Comparable & findMax( ) const
   {
       if( isEmpty( ) )
          throw UnderflowException{ };
       return findMax( root )->element;
   }
private:
    Node * root;
        //.private methods


};
```

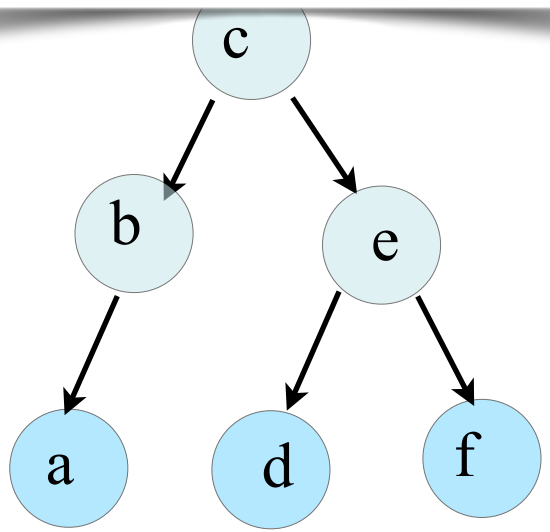
```
Node * findMax( Node *t ) const
{
   while( t->right != nullptr )
      t = t->right;

    return t;
}
```

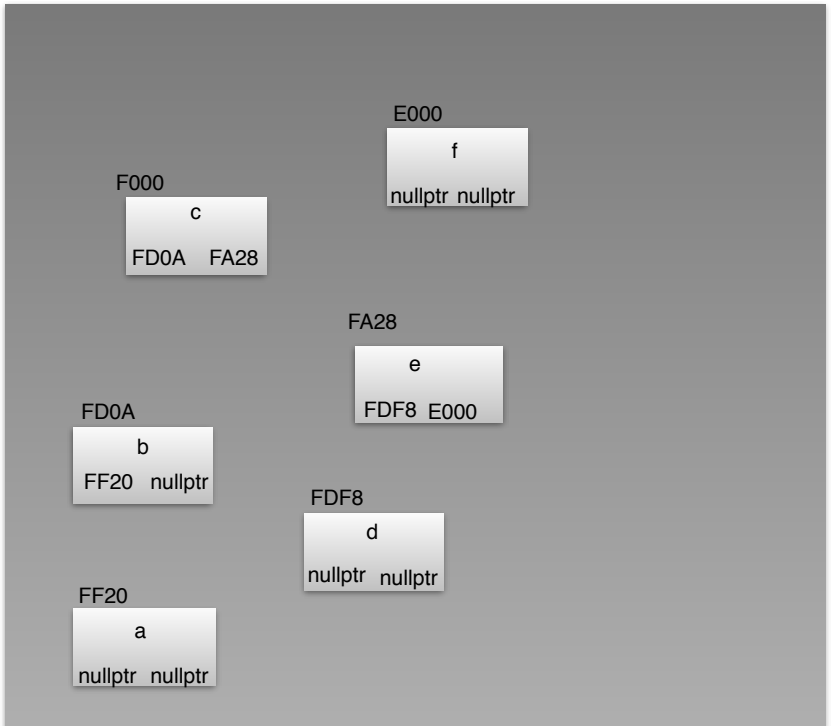
```
Internal method to find the largest item
in a subtree t.
Return node containing the largest item.
```



```
const Comparable & findMax( ) const
{
    if( isEmpty( ) )
       throw UnderflowException{ };
    return findMax( root )->element;
}
```

```
Node * findMax( Node *t ) const
{
    while( t->right != nullptr )
       t = t->right;

    return t;
}
```

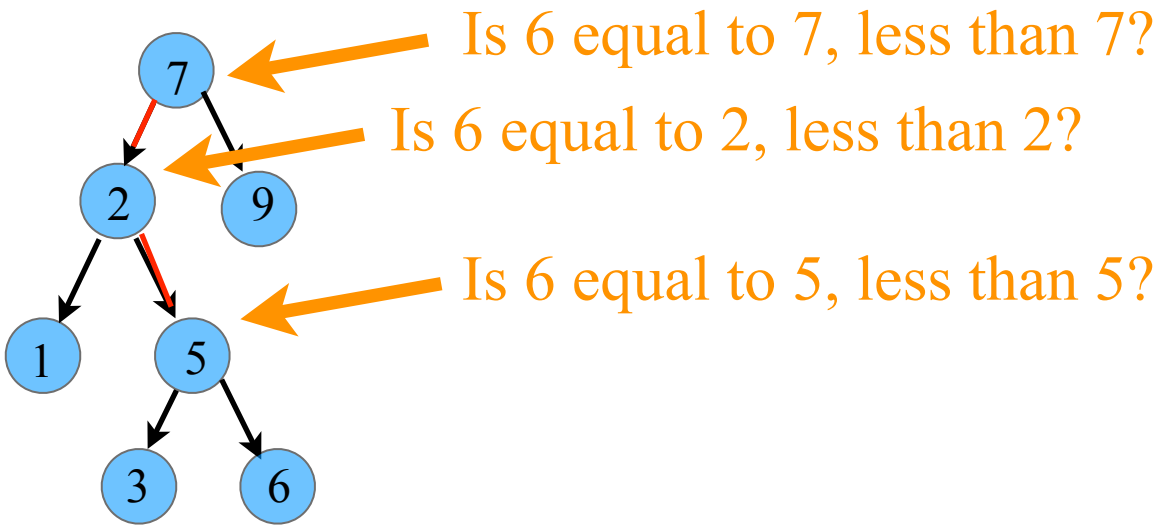




Insertion

Insert an object into the tree

If 6 is not in the tree, insert it into the tree



- Assume don't allow duplicate elements in tree
- Insertion is like search, but when reach NULL, insert new node there containing element
- Shape of tree depends on order of insertions

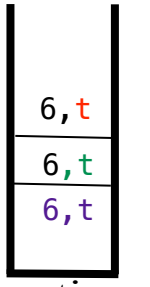
Inserting a new object into the binary search tree

```
template <class Comparable>
class BinarySearchTree
{
public:
    ...// public methods

    void insert( const Comparable & x )
   { insert( x, root ); }

private:
    Node * root;
    ...   // private methods
```

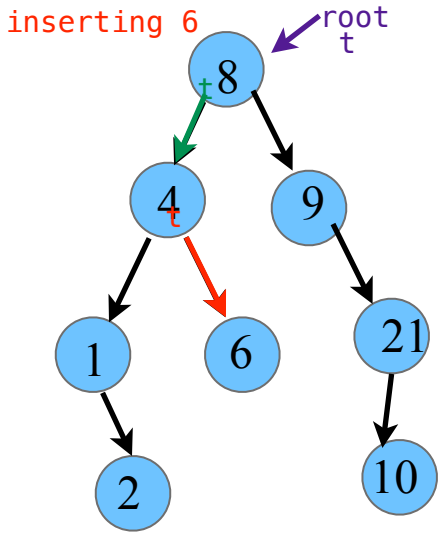
```
Internal method to insert into a subtree.
x is the item to insert.
t is the node that roots the tree.
Set the new root.
Throw DuplicateItemException if x is
```

```
void insert( const Comparable & x, Node * & t )
{
    if( t == nullptr )
        t = new Node( x, nullptr, nullptr, 0 );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        throw DuplicateItemException( );

    t->size++;
}
```

```
};
```

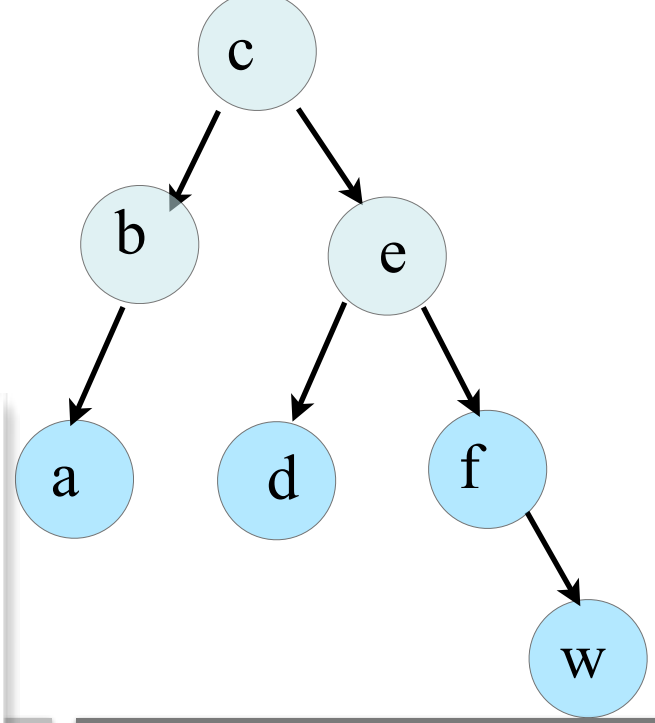


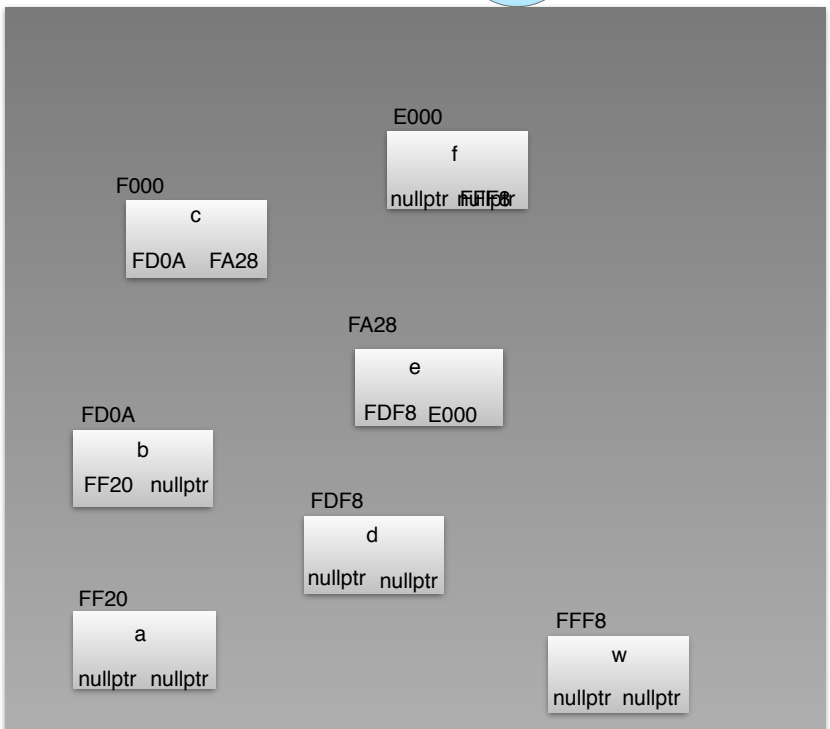
function call stack



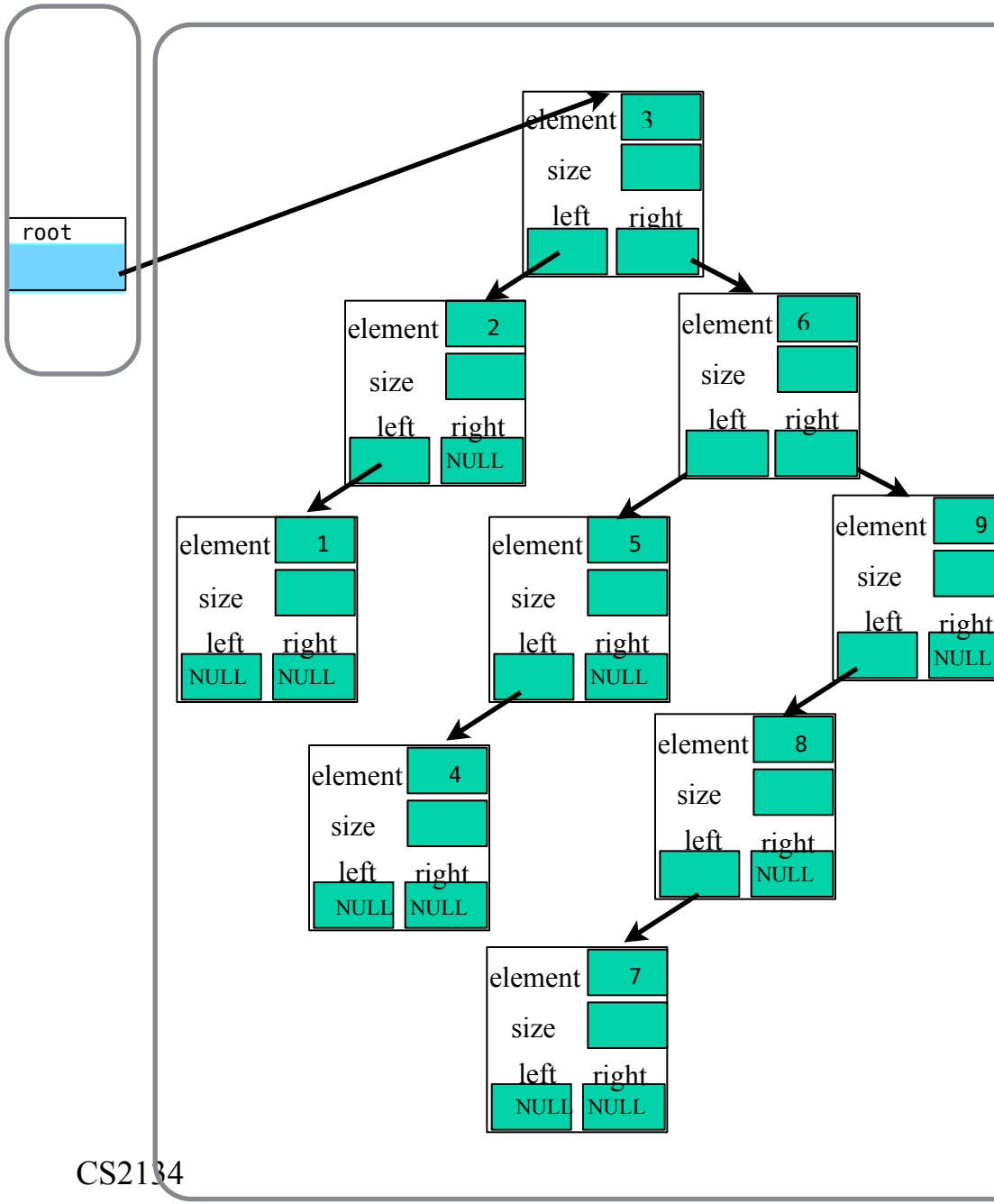
```
void insert( const Comparable & x )
{
    insert( x, root );
}
```

```
void insert( const Comparable & x, Node * & t )
{

    if( t == nullptr )
        t = new Node( x, nullptr, nullptr, 0 );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        throw DuplicateItemException( );

}   t->size++;
```



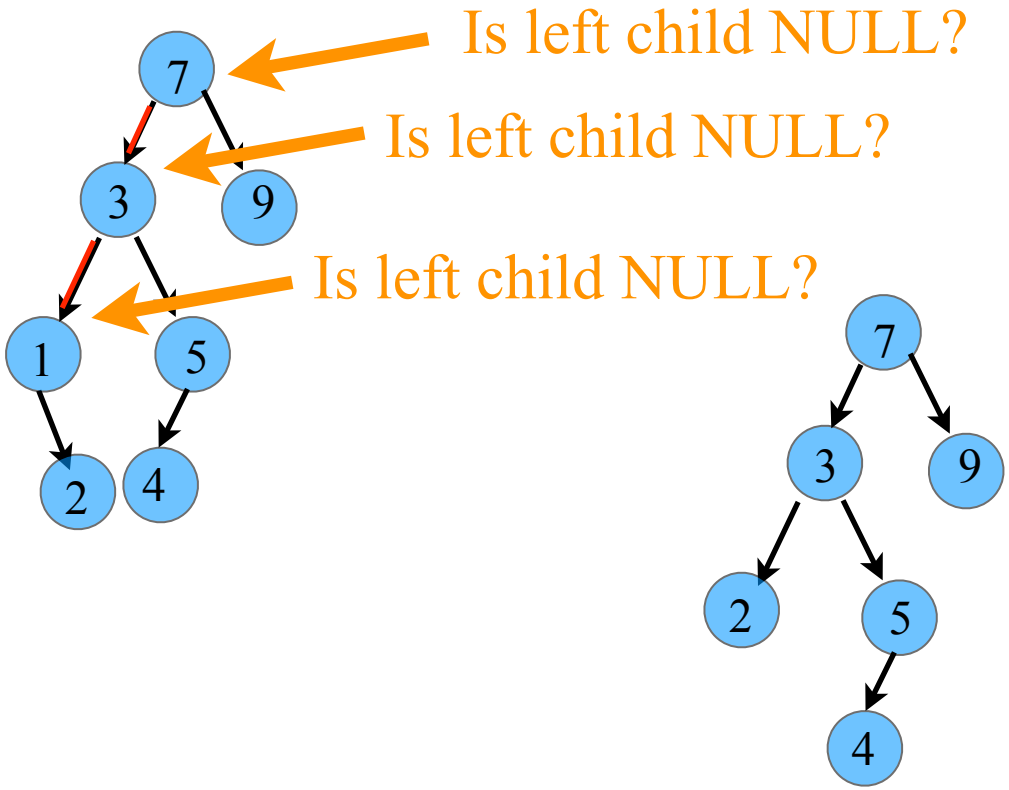


```
BinarySearchTree<int> t;
int NUMS = 10;
const int GAP  =  3;
int i;

for( i = GAP; i != 0; i = ( i + GAP ) % NUMS )
    t.insert( i );
for( i = 1; i < 10; i+= 2 )
        t.remove( i );
```



Deletion

Special Case: Deletion of Smallest object in the Tree



If the node is a leaf, delete the node and set parent's child pointer to NULL. Otherwise, attach the right child to the parent's node.

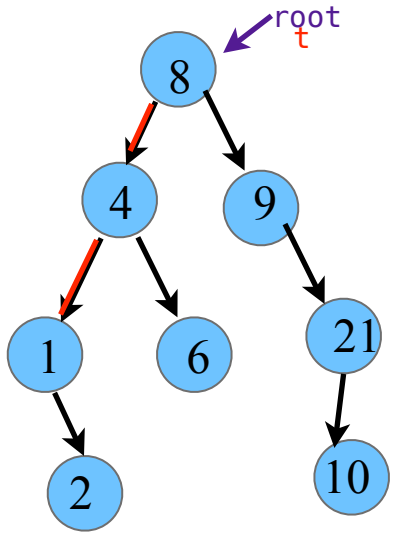
Removing the minimum object in the binary search tree

```
template <class Comparable>
class BinarySearchTree
{
public:
    ...// public methods

private:
    Node * root;

    void removeMin( ){ removeMin( root ); }

    ... // private methods
```

```
Internal method to remove minimum item from a subtree.
t is the node that roots the tree.
Set the new root.
Throw UnderflowException if t is empty.
```

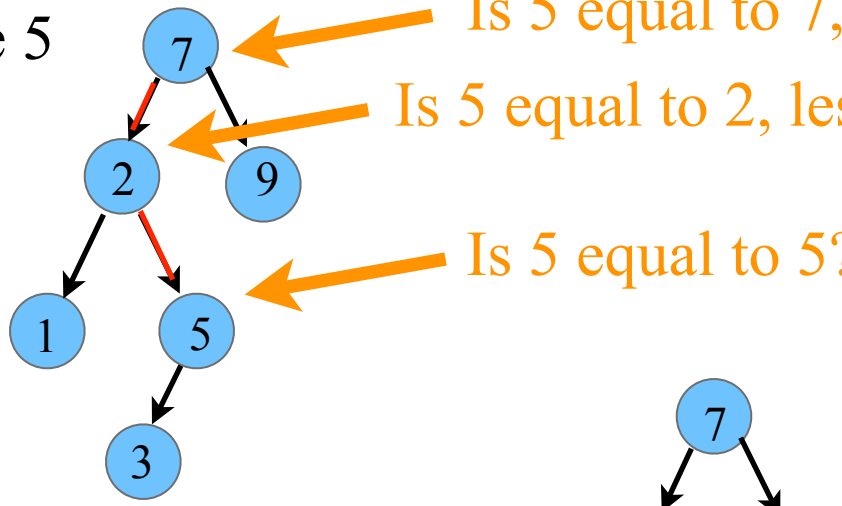
```
    void removeMin( Node * & t )
    {
        if( t == nullptr )
            throw UnderflowException( );
        else if( t->left != nullptr )
            removeMin( t->left );
        else
        {
            Node *tmp = t;
            t = t->right;
            delete tmp;
            return;
        }

        t->size--;
    }
```

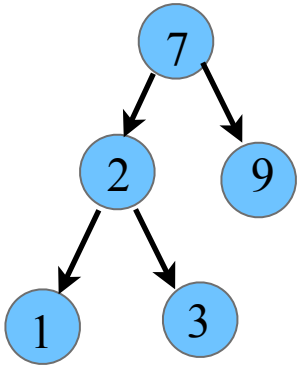
```
};
```



Deleting an internal node in the binary search tree

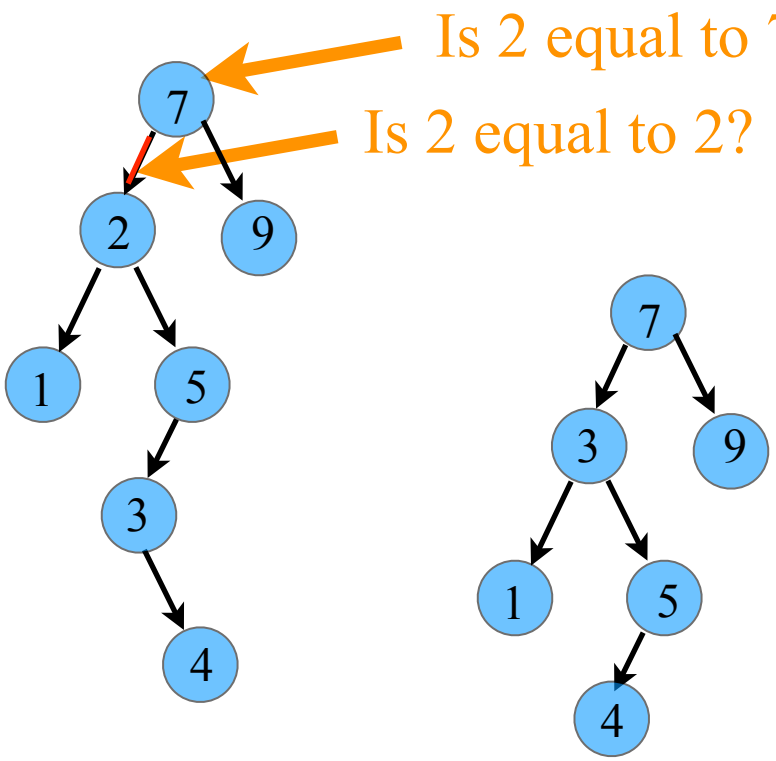
Delete 5





If the node has only one child - adjust parent's child link to bypass the node and then delete the node

Delete 2



If the node has two children, replace the node with the smallest item in right subtree.

Deletion

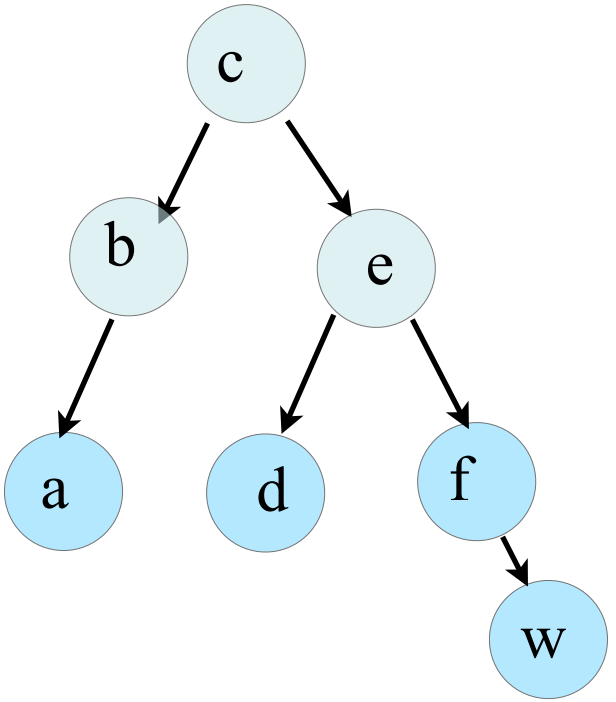
- Find node to delete
- If no children, remove (need to change parent)
- If one child, attach that child to node's parent
- If two children, replace node with its successor

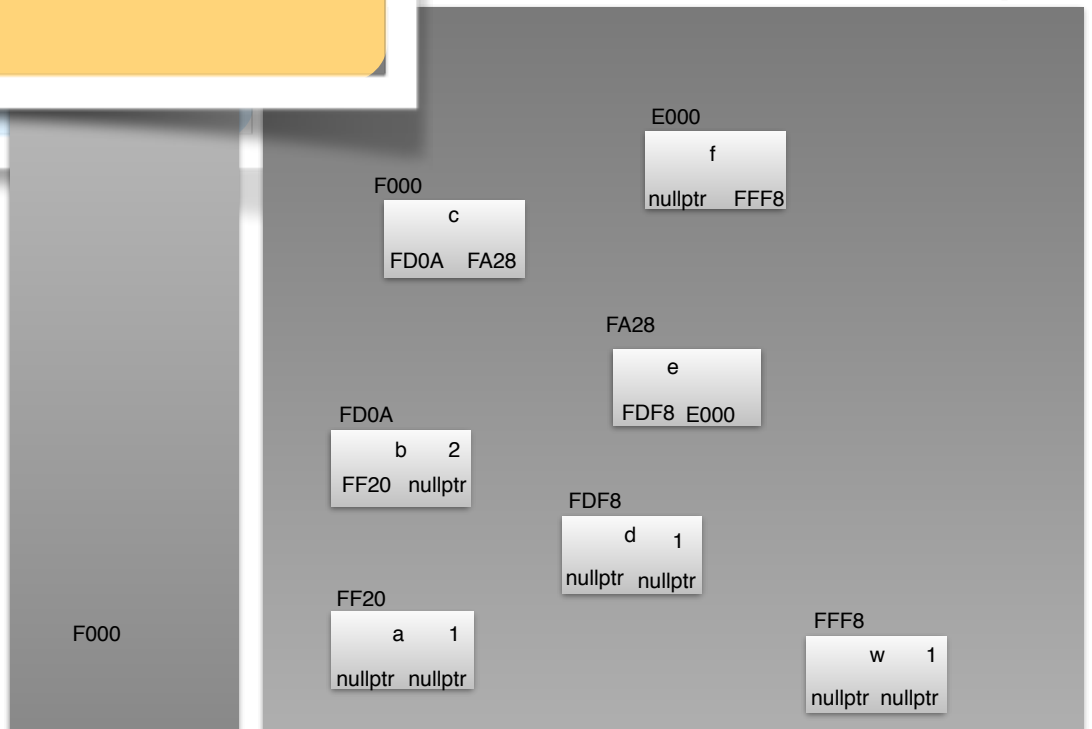
Removing an object in the binary search tree

```
template <class Comparable>
class BinarySearchTree
{
public:
    ...// public methods
    void remove( const Comparable & x ){ remove( x, root ); }
private:
    Node * root;
    ... // private methods

    /**
     * Internal method to remove from a subtree.
     * x is the item to remove.
     * t is the node that roots the tree.
     * Set the new root.
     * Throw ItemNotFoundException is x is not in t.
     */
    void remove( const Comparable & x, Node * & t )
    {
        if( t == nullptr )
            throw ItemNotFoundException( );
        if( x < t->element )
            remove( x, t->left );
        else if( t->element < x )
            remove( x, t->right );
        else if( t->left != nullptr && t->right != nullptr ){ // Two children
            t->element = std::move( findMin( t->right )->element );
            removeMin( t->right );                    // Remove minimum
        }
        else{
            BinaryNode<Comparable> *oldNode = t;
            t = ( t->left != nullptr ) ? t->left : t->right;  // Reroot t
            delete oldNode;                           // delete old root
            return;
        }
        t->size--;
    }
};
```

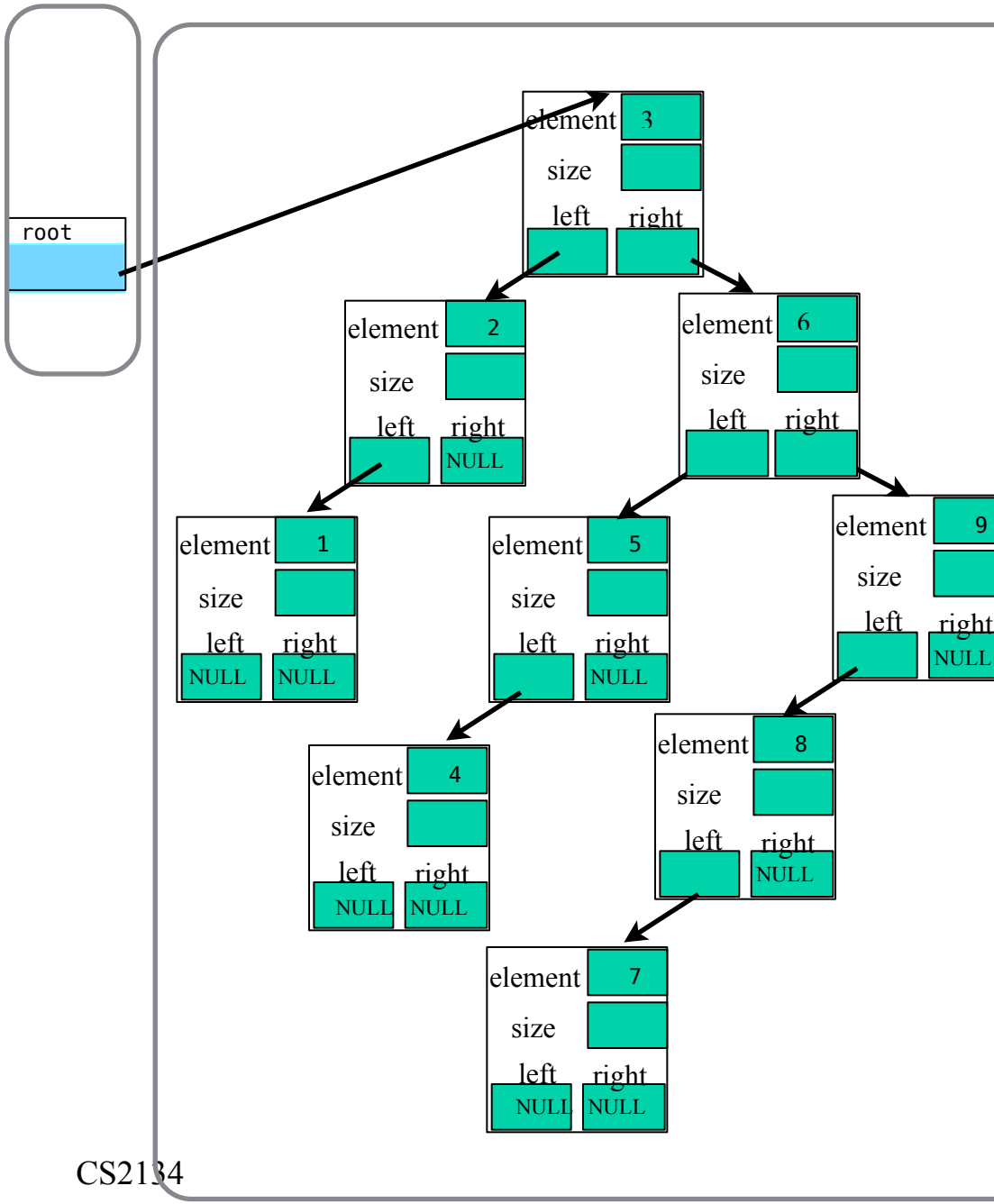
```
void remove( const Comparable & x ){ remove( x, root ); }
```

```
void remove( const Comparable & x, Node * & t )
{
  if( t == nullptr )
      throw ItemNotFoundException( );
  if( x < t->element )
      remove( x, t->left );
  else if( t->element < x )
      remove( x, t->right );
  else if( t->left != nullptr && t->right != nullptr ){ // Two children
      t->element = std::move( findMin( t->right )->element );
      removeMin( t->right );                    // Remove minimum
  }
  else{
      BinaryNode<Comparable> *oldNode = t;
      t = ( t->left != nullptr ) ? t->left : t->right;  // Reroot t
      delete oldNode;                           // delete old root
      return;
  }
  t->size--;
}
```

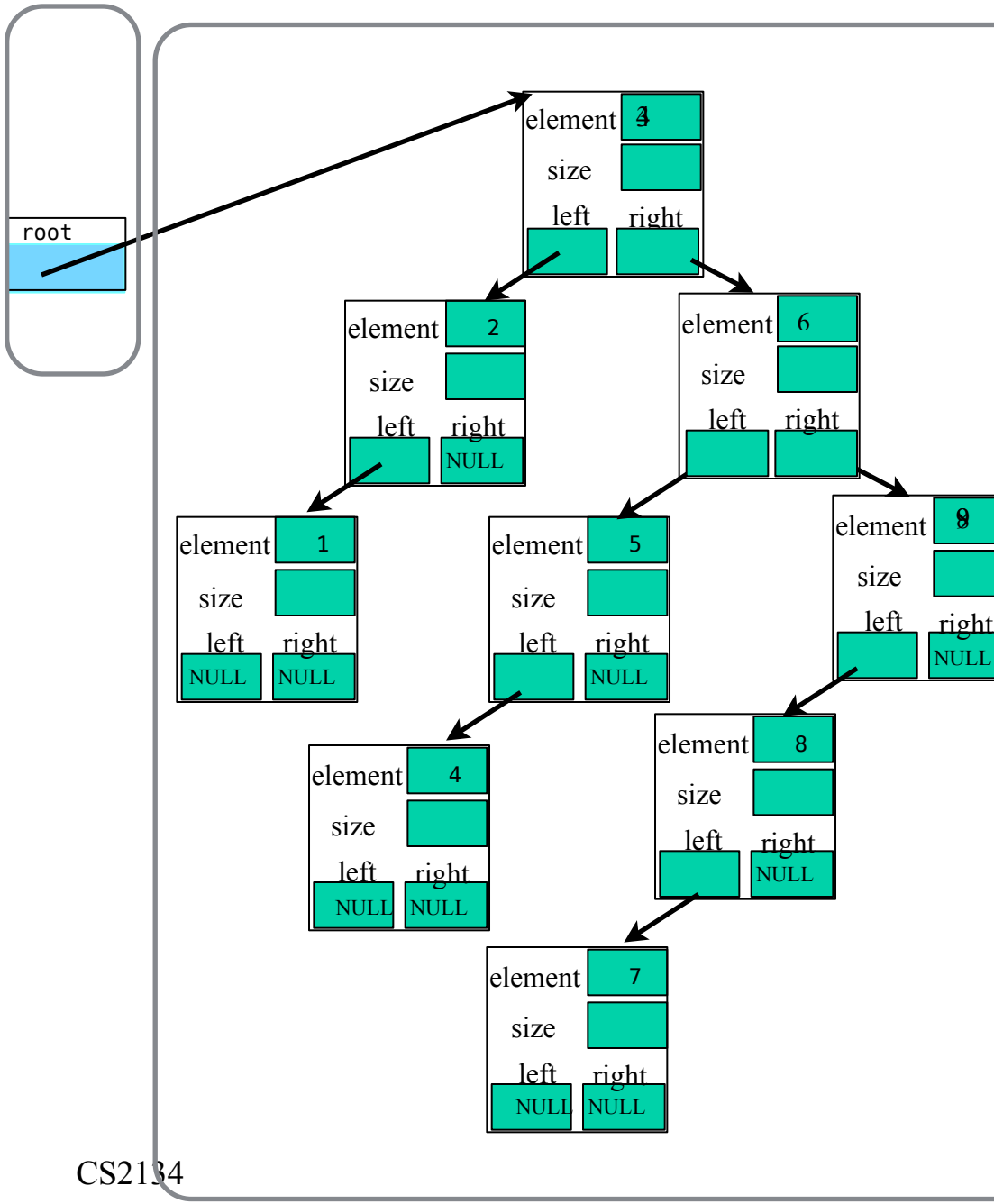




```
BinarySearchTree<int> t;
int NUMS = 10;
const int GAP  =  3;
int i;

for( i = GAP; i != 0; i = ( i + GAP ) % NUMS )
    t.insert( i );
```

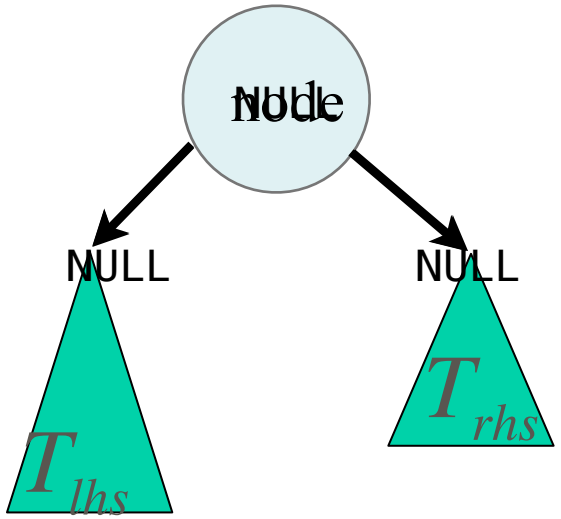


```
BinarySearchTree<int> t;
int NUMS = 10;
const int GAP  =  3;
int i;

for( i = GAP; i != 0; i = ( i + GAP ) % NUMS )
    t.insert( i );
for( i = 1; i < 10; i+= 2 )
        t.remove( i );
```



Make Empty

- If tree is empty - done
- Otherwise
  - Make left subtree empty
  - Make right subtree empty
  - Delete element
  - Set pointer to node to null



Removing all the nodes in an binary search tree

```
template <class Comparable>
class BinarySearchTree
{
public:
    ...// public methods
    void makeEmpty( ){ makeEmpty( root ); }
private:
    Node * root;
    ...  // private methods

   void makeEmpty( Node * & t )
   {
      if( t != nullptr )
      {
          makeEmpty( t->left );
          makeEmpty( t->right );
          delete t;
          t = nullptr;
      }
   }

};
```

Internal method to make subtree empty.

Copy Deep Copy

- If rhs is not the same lhs
  - Delete lhs
  - Set root to point to a new node where the
    - Left pointer points to a copy of the left subtree
    - Right pointer points to a copy of the right subtree
  - Update the size

The copy assignment operator for the binary search tree

```
template <class Comparable>
class BinarySearchTree
{
public:
    ...// public methods

  const BinarySearchTree & operator=( const BinarySearchTree & rhs )
  {
    if( this != &rhs )
     {
         makeEmpty( );
         root = clone( rhs.root );
     }
     return *this;
  }

  void makeEmpty( );

private:
    Node * root;
     ... // private methods
    Node * clone( Node *t ) const;
};
```

Deep copy.

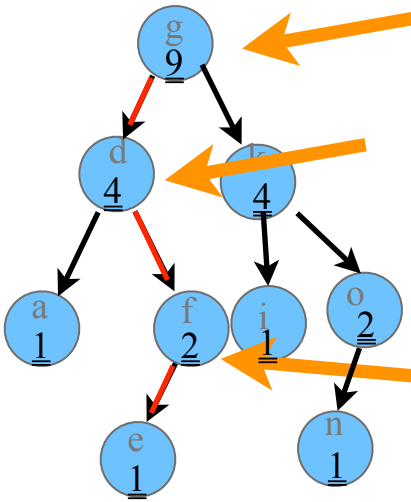
Copying the nodes in a binary tree

```
template <class Comparable>
class BinarySearchTree
{
public:
    ...// public methods


private:
    Node * root;
    ... // private methods

  Node * clone( Node * t ) const
  {
    if( t == nullptr )
        return nullptr;
    else
        return new Node( t->element, clone( t->left ), clone( t->right ), t->size );
  }


};
```

Internal method to clone subtree.

Special Search

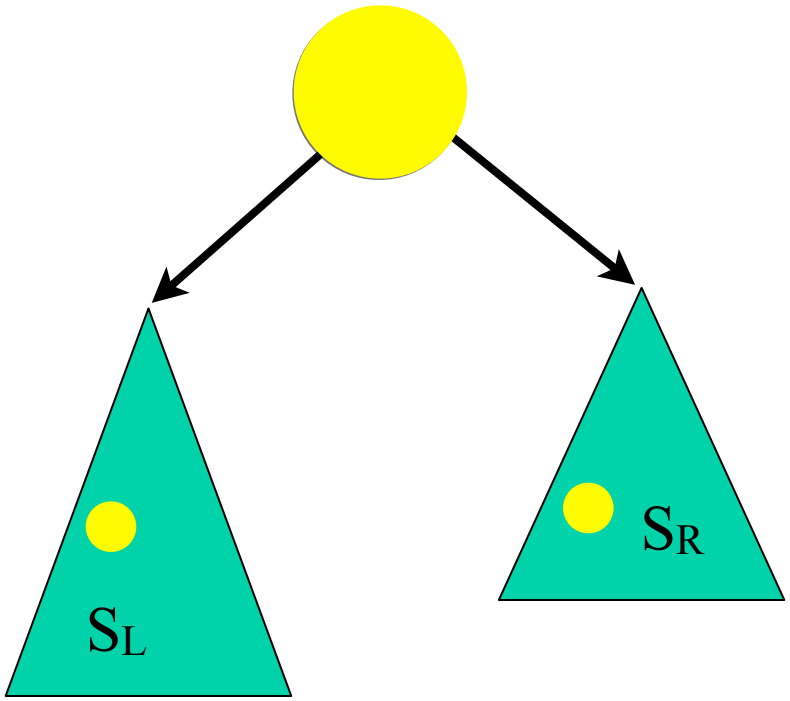
Finding the k'th smallest object in the binary search tree



Is 3rd smallest the root? In the left subtree?

Is 3rd smallest at this node? In the left subtree?

Is 3rd smallest at this node? In the left subtree?



If $K < S_L + 1$, the K^{th} smallest is in the left subtree!
If $K = S_L + 1$, the K^{th} smallest is the current node!
If $K > S_L + 1$, the K^{th} smallest is in the right subtree!
(The $(K- (S_L + 1))^{th}$ item in the right subtree.)

Finding the k'th smallest object in the binary search tree

```
template <class Comparable>
class BinarySearchTree
{
public:
     ...// public methods
     const Comparable &  findKth( int k ) const{
         if( isEmpty( ) || root->size < k)
               throw UnderflowException{ };
         return findKth( k, root )->element ;
      }
private:
    Node * root;
    ... // private methods

    Node * findKth( int k, Node * t ) const
    {


         int leftSize = treeSize( t->left );

         if( k <= leftSize )
             return findKth( k, t->left );
         else if( k == leftSize + 1 )
             return t;
         else
             return findKth( k - leftSize - 1, t->right );

    }

    int treeSize( Node *t ) const { return t == nullptr ? 0 : t->size; }
};
```

```
Internal method to find k'th item in a
subtree.
k is the desired rank.
t is the node that roots the tree.
```

What would the tree look like if
the objects were inserted in the order:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

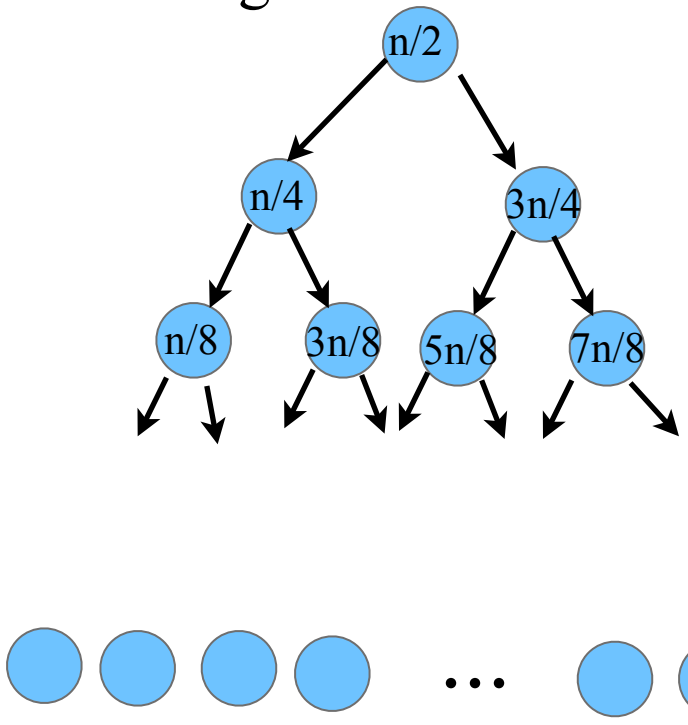
10, 9, 8, 7, 6, 5, 4, 3, 2, 1

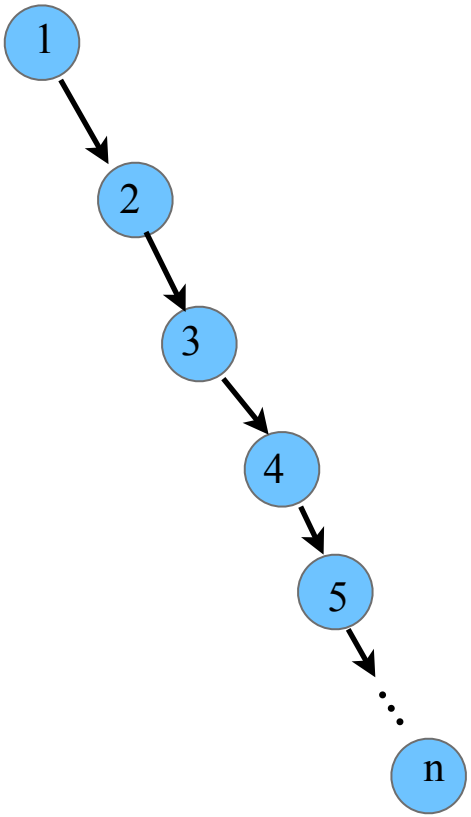
5, 3, 4, 2, 1, 7, 8, 9, 6, 10

How long does find take in the worst case?

Best case?

Average Case?





Analysis, using h and n

How long to traverse the binary search tree? O(n)

How long to find an object in the binary search tree? O(h)

How long to insert an object in the binary search tree? O(h)

How long to delete an object in the binary search tree? O(h)