Binary Search Tree

Binary Tree with value condition

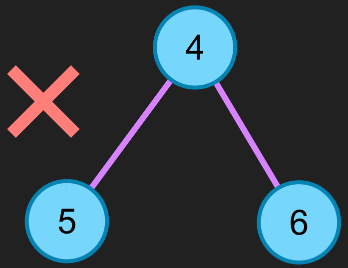
Nattee Niparnan

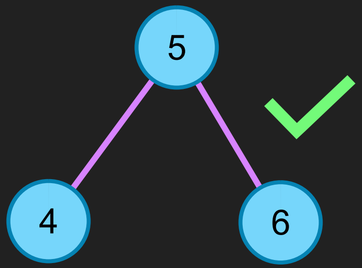
Overview

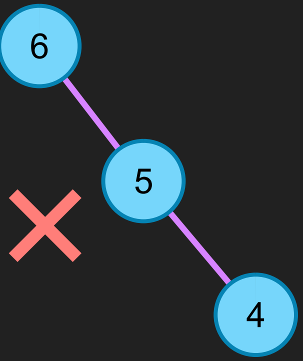
- We add additional value constraint to a Binary Tree
- The constraint make finding data in the tree much faster
  - $O(h)$ where h is the height of the tree
  - The tree is expected to have h be in $O(\lg n)$, but this is not always true
  - The next tree (AVL tree) will add more constraint so that we can guarantee that $h = O(\log n)$
- Using the same approach as a binary heap, maintain the constraint during modification

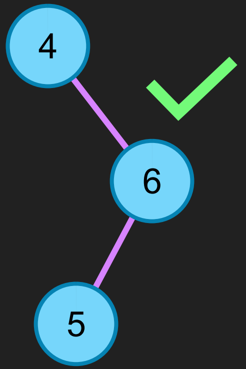
Binary Search Tree

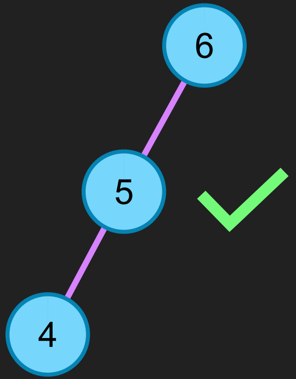
- Structure rule: must be a Binary Tree
- Value rule: for any node x
  - data in left-subtree must be less than the data in x
  - data in right-subtree must be more than the data in x
- Recursive Definition
  - An empty tree is a Binary Search Tree (BST)
  - A node X is a BST when
    - Its subtrees (if any) must be BST and
    - If left-subtree exists, X->data must be more than x->left->data
    - If right-subtree exists, X->data must be less than x->right->data





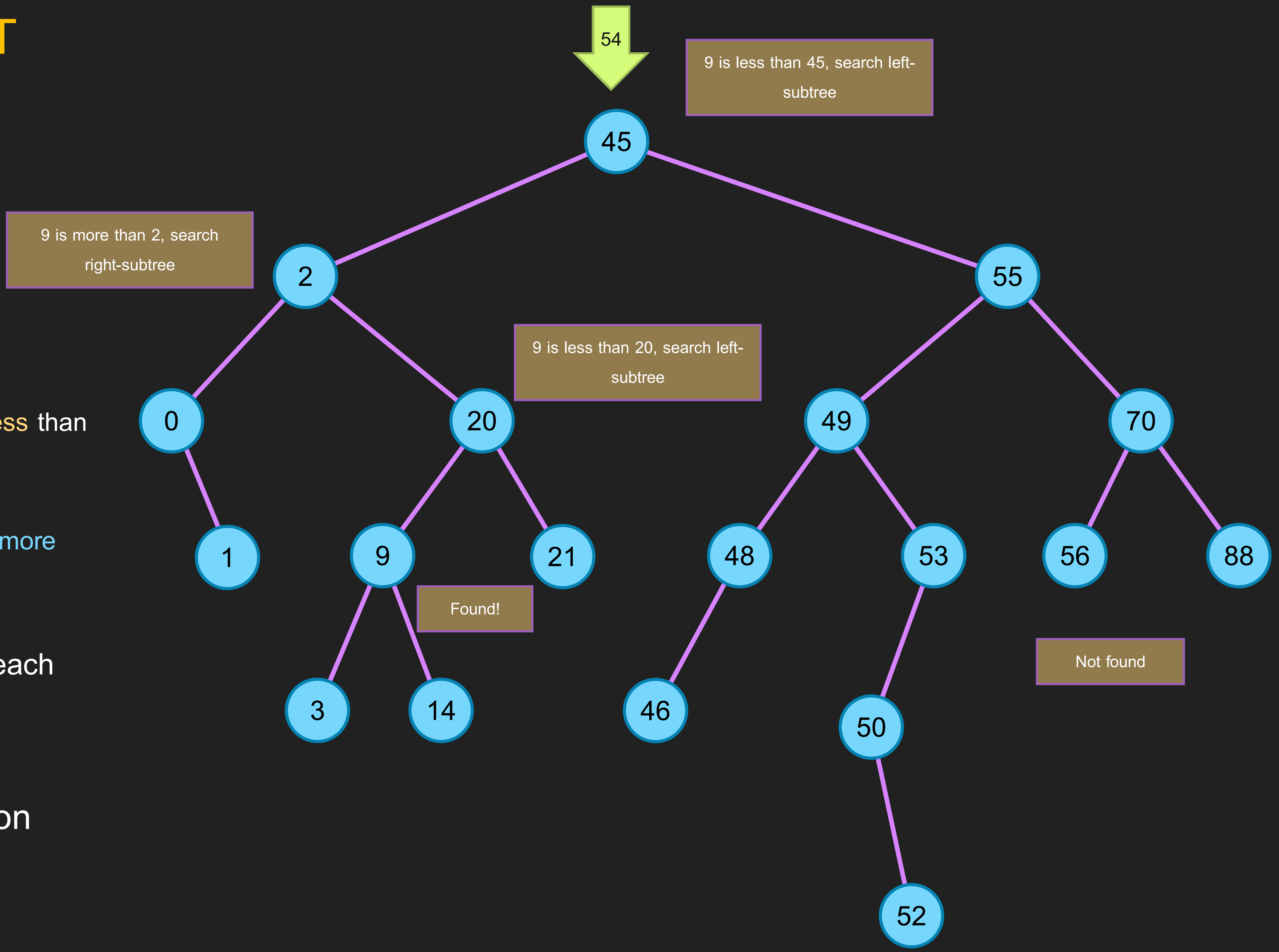






Finding Value in BST

- Value rules make finding fast
- To find e Start from root
  - If the current node is not e,
    - search in left-subtree if e is less than the current node
    - search in right-subtree if e is more than the current node
  - Keep going until we find e or reach NULL
- Other operation also depends on find



Find Node

Compare(a,b)

Return -1 if a < b

Return 0 if a == b

Return 1 if a > b

Later, we will need a parent node of the searching value

```
node* find_node(const ValueT& k,node* r, node* &parent) {
  node *ptr = r;
  while (ptr != NULL) {
    int cmp = compare(k, ptr->data);
    if (cmp == 0) return ptr;
    parent = ptr;
    ptr = cmp < 0 ? ptr->left : ptr->right;
  }
  return NULL;
}
```

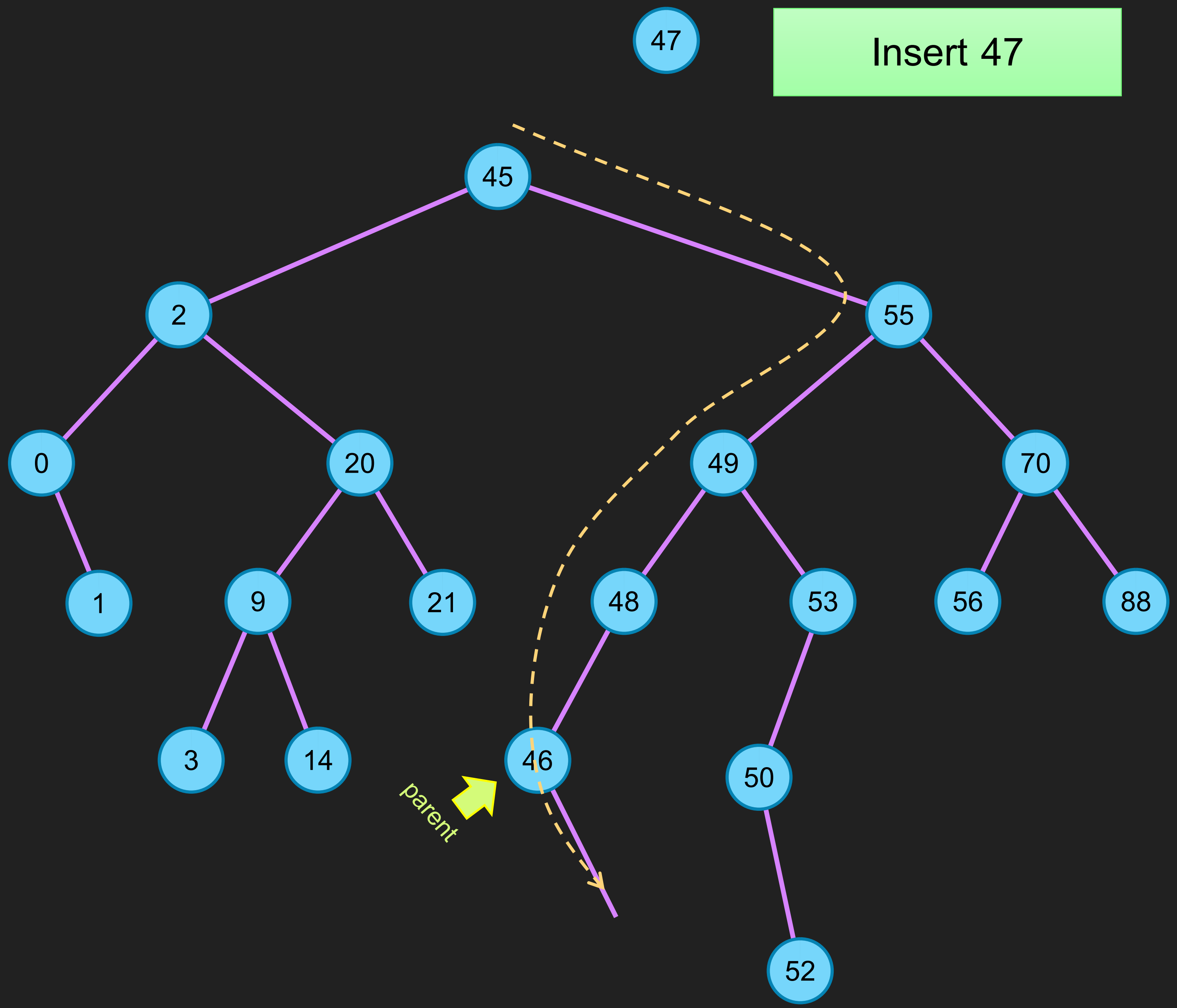
```
class node {
  friend class map_bst;
  protected:
    ValueT data;
    node  *left;
    node  *right;
    node  *parent;

    node() :
      data( ValueT() ), left( NULL ), right( NULL ), parent( NULL ) { }

    node(const ValueT& data, node* left, node* right, node* parent) :
      data ( data ), left( left ), right( right ), parent( parent ) { }
};
```

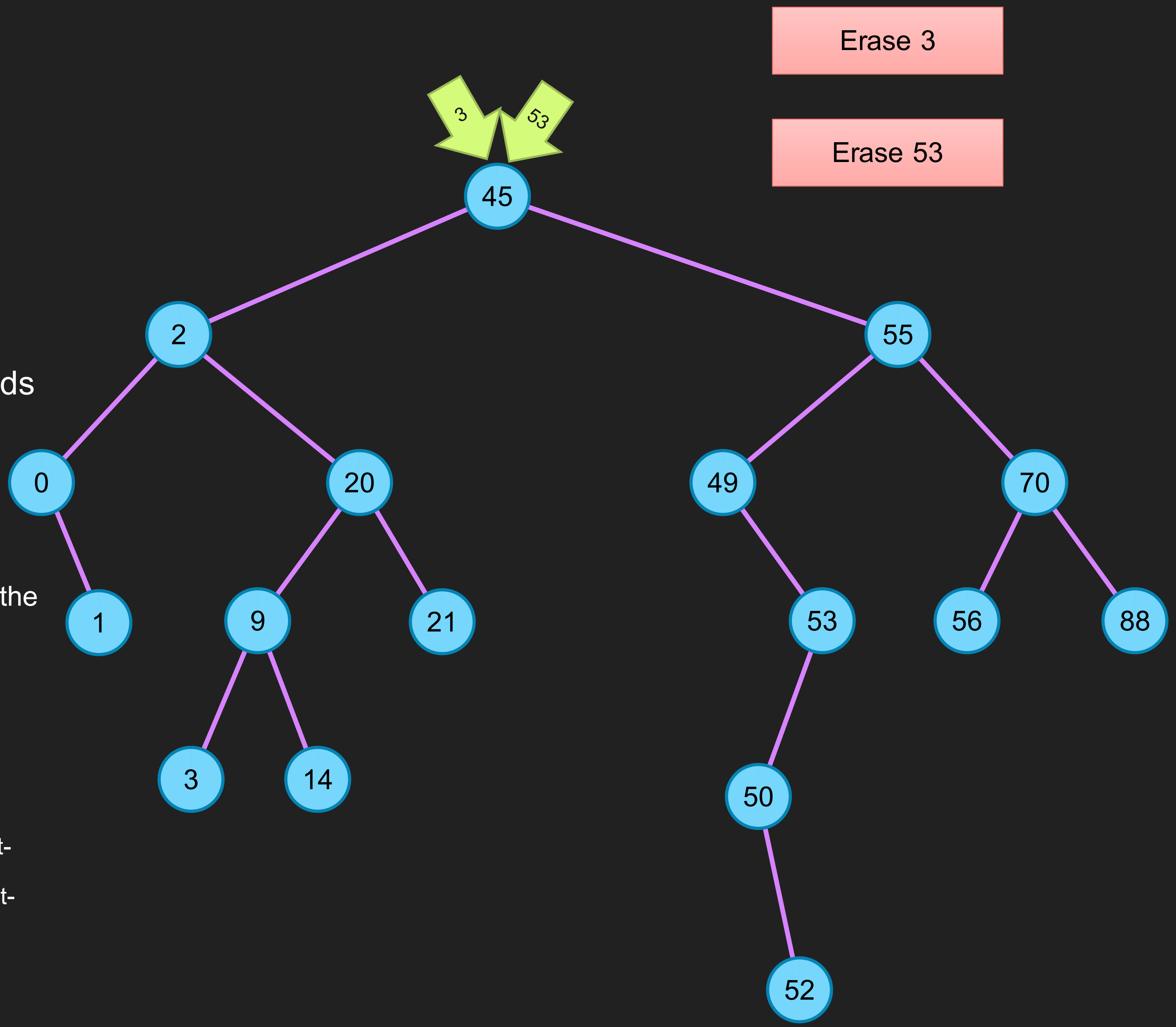
Insert

- Assumption: Data is BST is unique
- Insert(e) by find e
  - If e is found, don't add any node
  - If e is not in BST, find must reach NULL somewhere, that NULL is where to put e
- Both structure and value constraints are satisfied



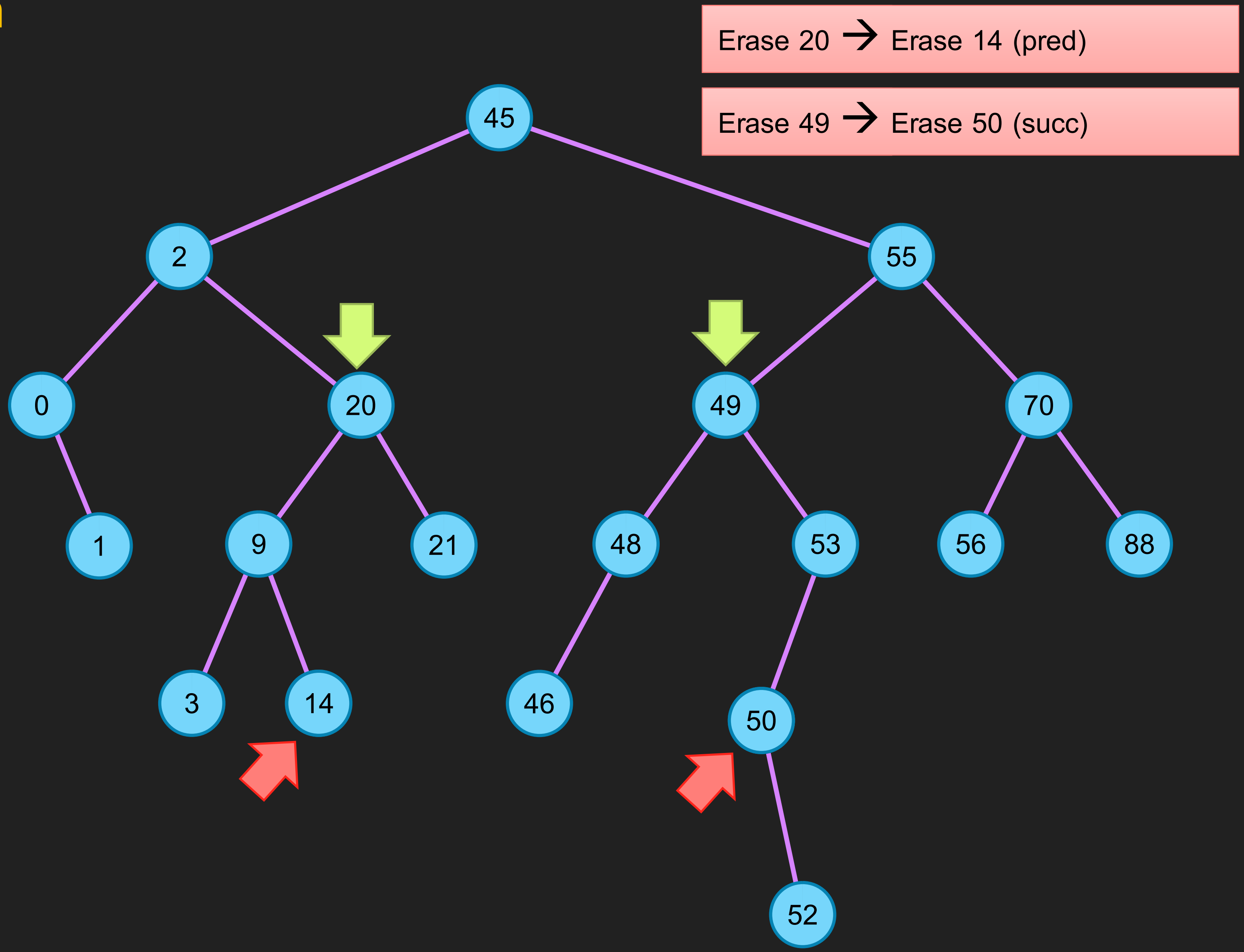
Erase

- erase(e) first have to find e as well
- If not found, do nothing
- If found at node X , there are 3 cases depends on number of children of e
  - If has no child, just simply delete X
  - If has one child, have parent of X points (using the same link) to the child of X instead
  - If has two children, pick either successor or predecessor of e
    - Assume we choose successor p (must be in right-subtree), replace X with P and erase(p) from right-subtree



Erase node with 2 children

- Replace by successor (or predecessor) preserves value rules
  - Successor is the minimum in right subtree
  - Predecessor is the maximum in left subtree
- Both exists (because the node has both subtrees)



Finding Successor and Predecessor

- Successor is the minimum in right-subtree
- If a tree has left-subtree, min is the min of left-subtree
  - If not, min is the root

- Predecessor is the maximum in left-subtree
- If a tree has right-subtree, max is the max of right-subtree
  - If not, max is the root

```
node* find_min_node(node* r) {
  //r must not be NULL
  node *min = r;
  while (min->left != NULL) {
    min = min->left;
  }
  return min;
}
```

```
node* find_max_node(node* r) {
  //r must not be NULL
  node *max = r;
  while (max->right != NULL) {
    max = max->right;
  }
  return max;
}
```

Finding Successor and Predecessor (recursive)

- Successor is the minimum in right-subtree
- If a tree has left-subtree, min is the min of left-subtree
  - If not, min is the root

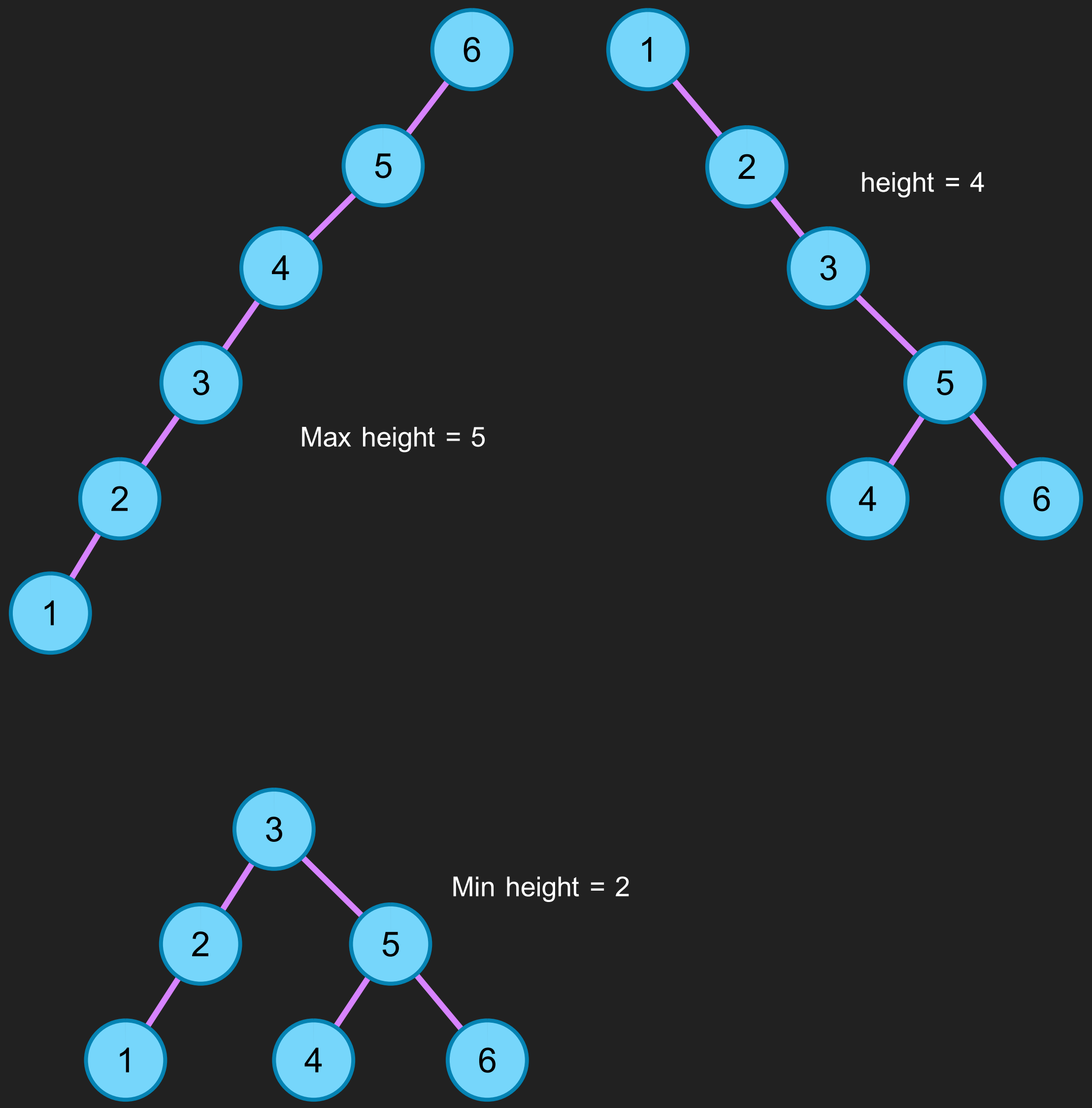
```
node* find_min_node(node* r) {
  //r must not be NULL
  if (r->left == NULL) return r;
  return find_min_node(r->left);
}
```

- Predecessor is the maximum in left-subtree
- If a tree has right-subtree, max is the max of right-subtree
  - If not, max is the root

```
node* find_max_node(node* r) {
  //r must not be NULL
  if (r->right == NULL) return r;
  return find_max_node(r->right);
}
```

Complexity Analysis

- Insert, erase depends in find, find_min (or find_max)
- All finds start from root and in the worst case reach the leaf
    - Hence, O(h)
- Height of the tree can be in the range from n to lg n
- For 1,000,000 nodes, its in the range of [20,999999]
    - O(h) is, right now, O(n)
    - Will be fixed by AVL tree



CP::map_bst

Using Binary Search Tree to create associated data structure

Layout

- Need node class
- Also need iterator class
- Template has two types
  - Key Type and Mapped Type
  - ValueType is

    pair<KeyType,MappedType>
- Also need custom comparator

```
template <typename KeyT,
          typename MappedT,
          typename CompareT = std::less<KeyT> >
class map_bst {
  protected:
    typedef std::pair<KeyT,MappedT> ValueT;
    class node {
      friend class map_bst;
      protected:
        ValueT data;
        node  *left;
        node  *right;
        node  *parent;
    };
    class tree_iterator {
      protected:
        node* ptr;
      public:
    };
    node     *mRoot;
    CompareT  mLess;
    size_t    mSize;
  public:
    typedef tree_iterator iterator;
};
```

Node Class

- Data stores both the key type and mapped type (as a pair)
- Map finds by key

```
class node {
  friend class map_bst;
  protected:
    ValueT data;
    node   *left;
    node   *right;
    node   *parent;

    node() :
      data( ValueT() ), left( NULL ), right( NULL ), parent( NULL ) { }

    node(const ValueT& data, node* left, node* right, node* parent) :
      data ( data ), left( left ), right( right ), parent( parent ) { }
};
```

Ctors, Dtor

```
map_bst(const map_bst<KeyT,MappedT,CompareT> & other) :
  mLess(other.mLess) , mSize(other.mSize)
{ mRoot = copy(other.mRoot, NULL); }

map_bst(const CompareT& c = CompareT() ) :
  mRoot(NULL), mLess(c) , mSize(0)
{ }

map_bst<KeyT,MappedT,CompareT>& operator=(map_bst<KeyT,MappedT,CompareT> other)  {
  using std::swap;
  swap(this->mRoot, other.mRoot);
  swap(this->mLess, other.mLess);
  swap(this->mSize, other.mSize);
  return *this;
}

~map_bst() {
  clear();
}
```

Recursive Copy

Recursive delete

Actual Find

- Find by Key

```
iterator find(const KeyT &key) {
  node *parent;
  node *ptr = find_node(key,mRoot,parent);
  return ptr == NULL ? end() : iterator(ptr);
}
```

```
int compare(const KeyT& k1, const KeyT& k2) {
  if (mLess(k1, k2)) return -1;
  if (mLess(k2, k1)) return +1;
  return 0;
}
node* find_node(const KeyT& k,node* r, node* &parent) {
  node *ptr = r;
  while (ptr != NULL) {
    int cmp = compare(k, ptr->data.first);
    if (cmp == 0) return ptr;
    parent = ptr;
    ptr = cmp < 0 ? ptr->left : ptr->right;
  }
  return NULL;
}
```

Insert

- Insert return pair of iterator

  and insert result

```
node* &child_link(node* parent, const KeyT& k)
{
    if (parent == NULL) return mRoot;
    return mLess(k, parent->data.first) ?
                parent->left : parent->right;
}
```

```
std::pair<iterator,bool> insert(const ValueT& val) {
  node *parent = NULL;
  node *ptr = find_node(val.first,mRoot,parent);
  bool not_found = (ptr==NULL);
  if (not_found) {
    ptr = new node(val,NULL,NULL,parent);
    child_link(parent, val.first) = ptr;
    mSize++;
  }
  return std::make_pair(iterator(ptr), not_found);
}
```

child_link return a reference (the variable)

to the pointer of the appropriate child of

the parent with respect to k

Erase

```
size_t erase(const KeyT &key) {
  if (mRoot == NULL) return 0;
  node *parent = NULL;
  node *ptr = find_node(key,mRoot,parent);
  if (ptr == NULL) return 0;
  if (ptr->left != NULL && ptr->right != NULL) {
    //have two children
    node *min = find_min_node(ptr->right);
    node * &link = child_link(min->parent, min->data.first);
    link = (min->left == NULL) ? min->right : min->left;
    if (link != NULL) link->parent = min->parent;
    std::swap(ptr->data.first, min->data.first);
    std::swap(ptr->data.second, min->data.second);
    ptr = min; // we are going to delete this node instead
  } else {
    node * &link = child_link(ptr->parent, key);
    link = (ptr->left == NULL) ? ptr->right : ptr->left;
    if (link != NULL) link->parent = ptr->parent;
  }
  delete ptr;
  mSize--;
  return 1;
}
```

- Handle multiple cases

Operator[ ]

```
MappedT& operator[](const KeyT& key) {
  node *parent = NULL;
  node *ptr = find_node(key, mRoot, parent);
  if (ptr == NULL) {
    ptr = new node(std::make_pair(key,MappedT()),NULL,NULL,parent);
    child_link(parent, key) = ptr;
    mSize++;
  }
  return ptr->data.second;
}
```

- Find node
- If not exists, create one with default MappedTypeReturn MappedType of the node

Iterator

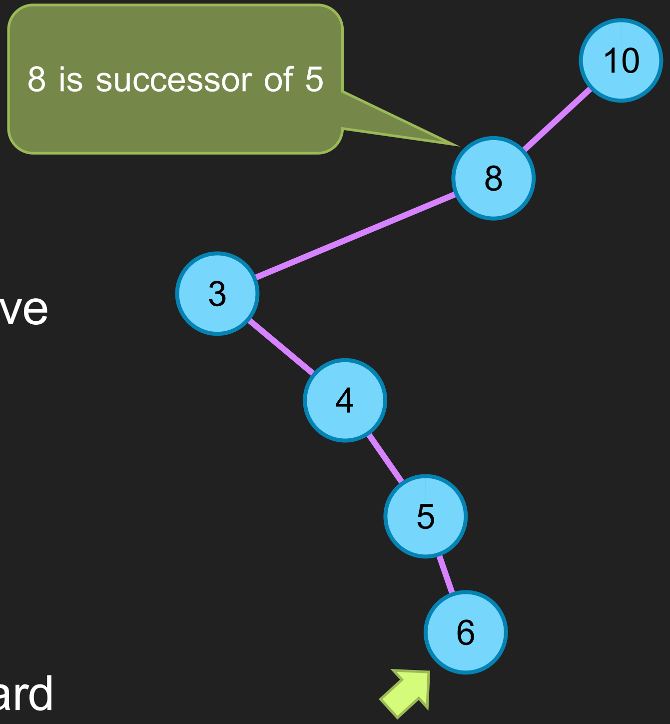
- Just like linked list, we need a class for iterator
    - Because we need custom operator++, -- (and some more)
- Iterator class just store a pointer to a node

```
class tree_iterator {
  protected:
    node* ptr;

  public:
    tree_iterator() : ptr( NULL ) { }
    tree_iterator(node *a) : ptr(a) { }
    // more functions below
};
```

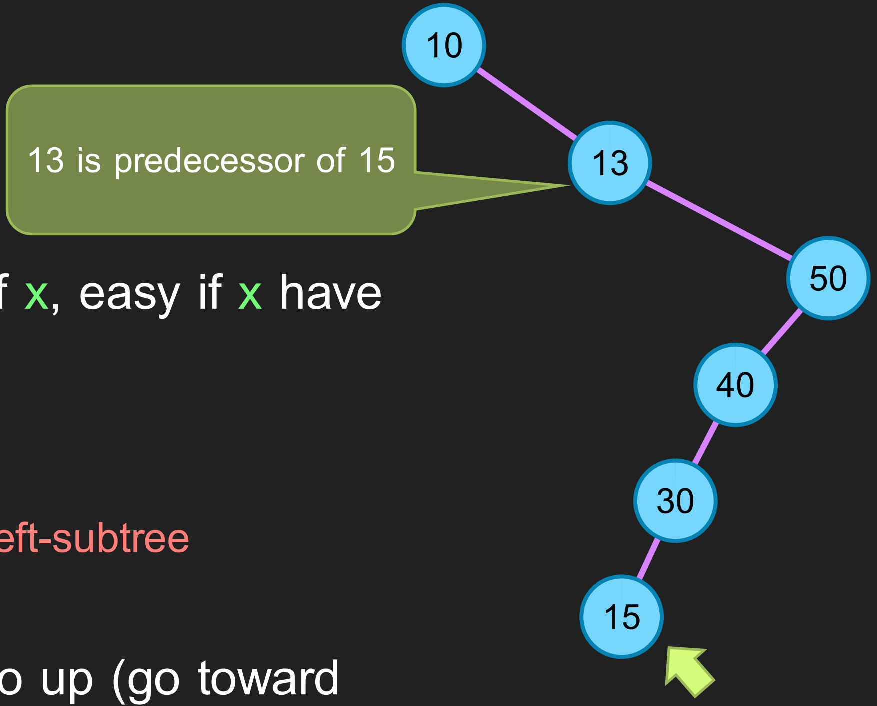
Operator++

- Find successor of x, easy if x have right-subtree
  - Just find min of right-subtree
- If not, we have to go up (go toward root) until we find one that is more than x
  - This is always the closest ancestor of x that has x in its left-subtree




```
tree_iterator& operator++() {
  if (ptr->right == NULL) {
    node *parent = ptr->parent;
    while (parent != NULL &&
           parent->right == ptr) {
      ptr = parent;
      parent = ptr->parent;
    }
    ptr = parent;
  } else {
    ptr = ptr->right;
    while (ptr->left != NULL)
      ptr = ptr->left;
  }
  return (*this);
}
```

Operator--



- Find predecessor of x, easy if x have left-subtree
    - Just find max of left-subtree
- If not, we have to go up (go toward root) until we find one that is less than x
    - This is always the closest ancestor of x that has x in its right-subtree

```
tree_iterator& operator--() {
  if (ptr->left == NULL) {
    node *parent = ptr->parent;
    while (parent != NULL &&
           parent->left == ptr) {
      ptr = parent;
      parent = ptr->parent;
    }
    ptr = parent;
  } else {
    ptr = ptr->left;
    while (ptr->right != NULL)
      ptr = ptr->right;
  }
  return (*this);
}
```

Other Functions

```
tree_iterator operator++(int) {
  tree_iterator tmp(*this);
  operator++();
  return tmp;
}

tree_iterator operator--(int) {
  tree_iterator tmp(*this);
  operator--();
  return tmp;
}

ValueT& operator*()  { return ptr->data;    }
ValueT* operator->() { return &(ptr->data); }
bool    operator==(const tree_iterator& other)
  { return other.ptr == ptr; }
bool    operator!=(const tree_iterator& other)
  { return other.ptr != ptr; }
```

Summary

- Binary Search Tree relies on Value Constraint to make find fast
  - Possible to be slow (will be fixed later)
- Erase requires find_min, max
- CP::map_bst use pair to store KeyT and MappedT
  - Find use Key
- Iterator is just a pointer
  - Have a problem of operator-- at end() (will be fixed later)