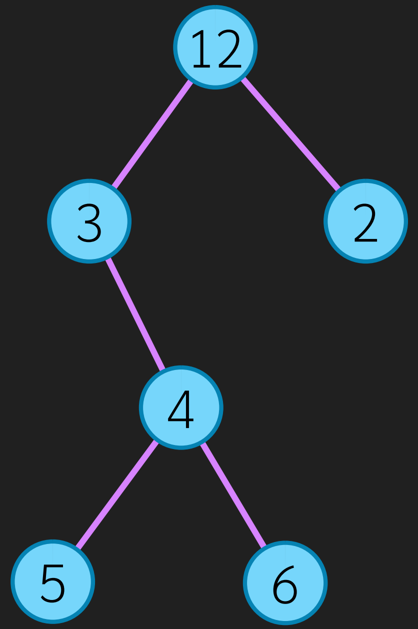
Binary Tree

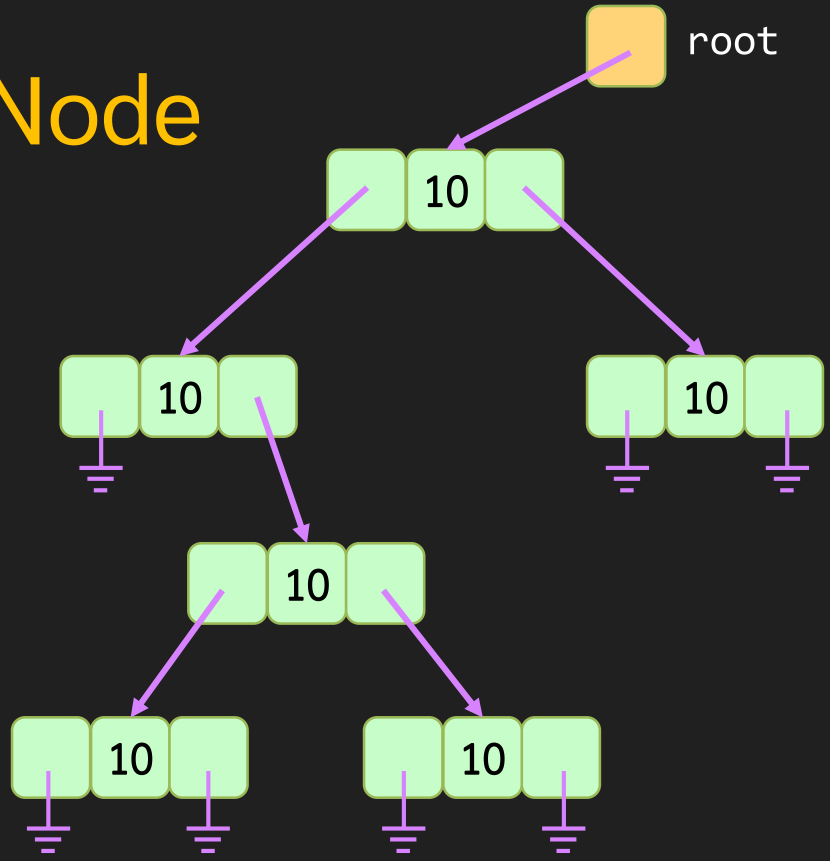
Practicing Pointer & Recursive

Overview

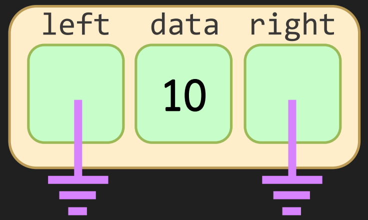
- This is a basic for the next data structure, Binary Search and AVL Tree
- Focus on using Node and Pointer
- Focus on using recursive programming
- Some applications using just Binary Tree
- There is no data structure in std that is Binary Tree

Binary Tree & Node



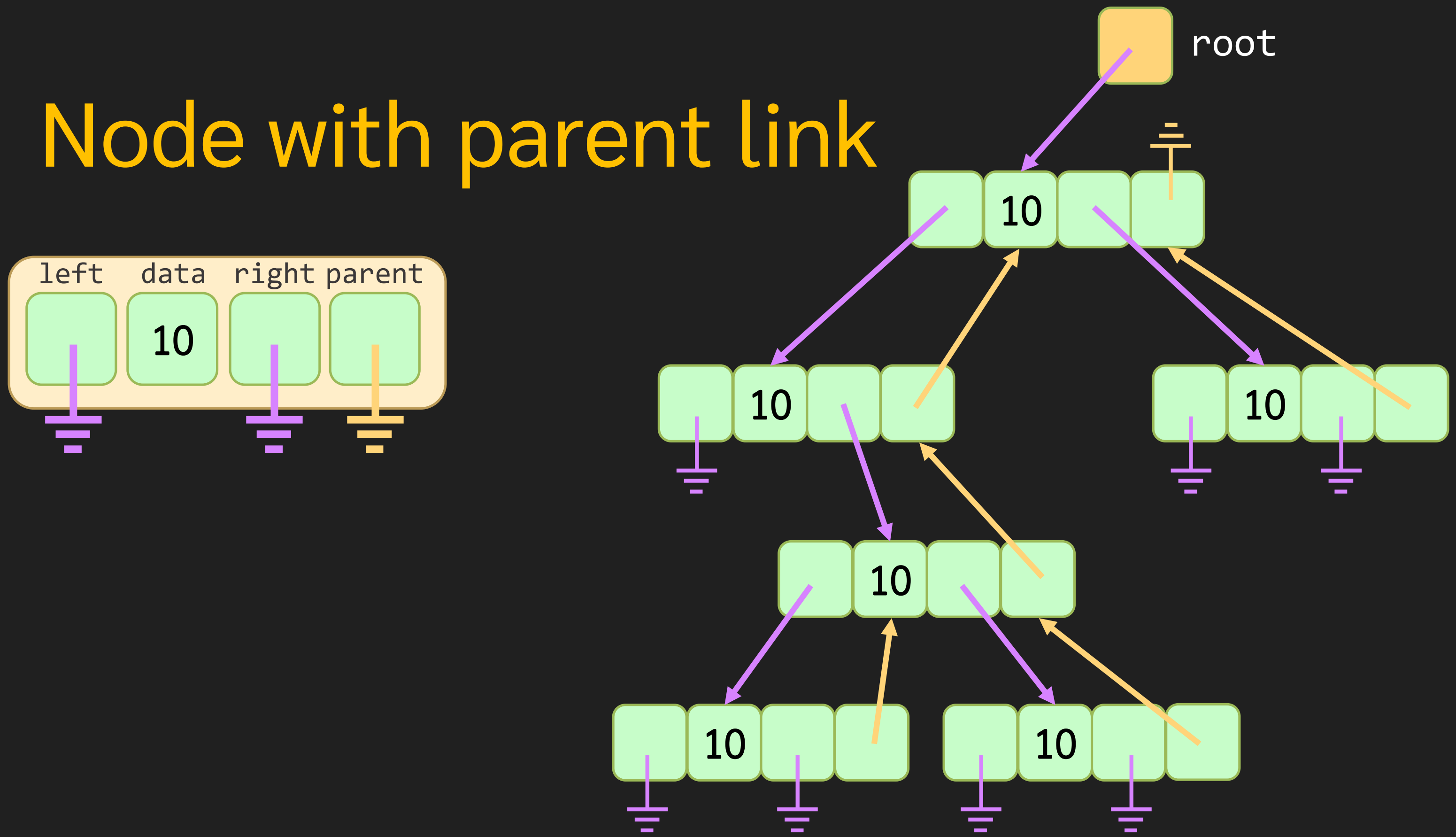


- A rooted tree where each node have at most two children
- Tree Node is very similar to a linked list node




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

Node with parent link



- Sometime, we need a link to parent
- Root is the only node that parent is NULL

```
class node {
  public:
    ValueT data;
    node  *left, *right, *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 ) { }
};
```

Huffman Coding: Example Application of Tree

- David Huffman proposed this as his term project in Robert Fano’s class (co-worker of Claude Shannon) which beats Shannon-Fano encoding
- Encoding = associate meaning to a representation
- ASCII Code
  - Fix length encoding
  - Each char = 8 bits

100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M

Variable Length Encoding

Never gonna give you up
Never gonna let you down
Never gonna run around and desert you

16 different character
Fix-length needs 4 x 86 = 344 bits
Variable Length need 327 bits

n	e	o	u	r	a	v	g	d	y	t	w	s	p	l	i
14	11	9	7	7	6	5	5	5	4	3	2	2	2	2	2
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
11	010	011	0001	0011	0000	1011	1010	1000	00101	10011	100101	0010001	001001	100100	0010000

Encoding “Never”
Fix-length 00000001011000010100
Variable Legnth 11010101101000011

Problem Statement

ความยาว ไม่เท่ากัน

b: $\boxed{110}$
c: 001
d: 010

- Input: $\boxed{\text{a string}}$
- Output: encoding of each character in the string such that

huffman

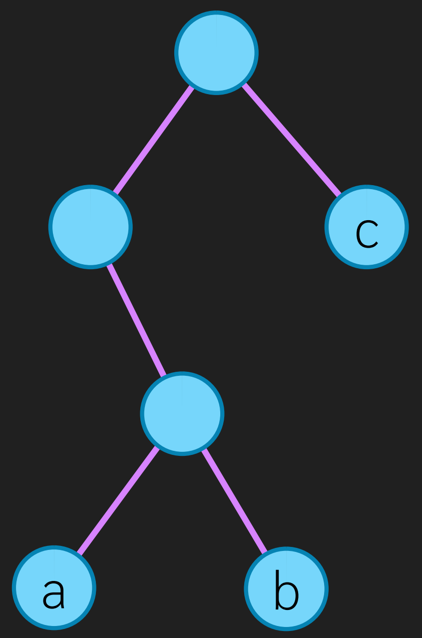
- The total length of encoding the string is minimum → algo design

2

- The encoding of character is not ambiguous
    - Any character encoding is not a prefix of any other character

Tree Encoding

- Using a tree to represent encoding
- Each character is represent at leaf nodes
  - Leaf node is a node without children
- Encode by start at the root and walk toward leaf nodes
  - The path gives the encoding
  - Going to left child equal to 0
  - Going to right child equal to 1
- Guarantee to be non-ambiguous



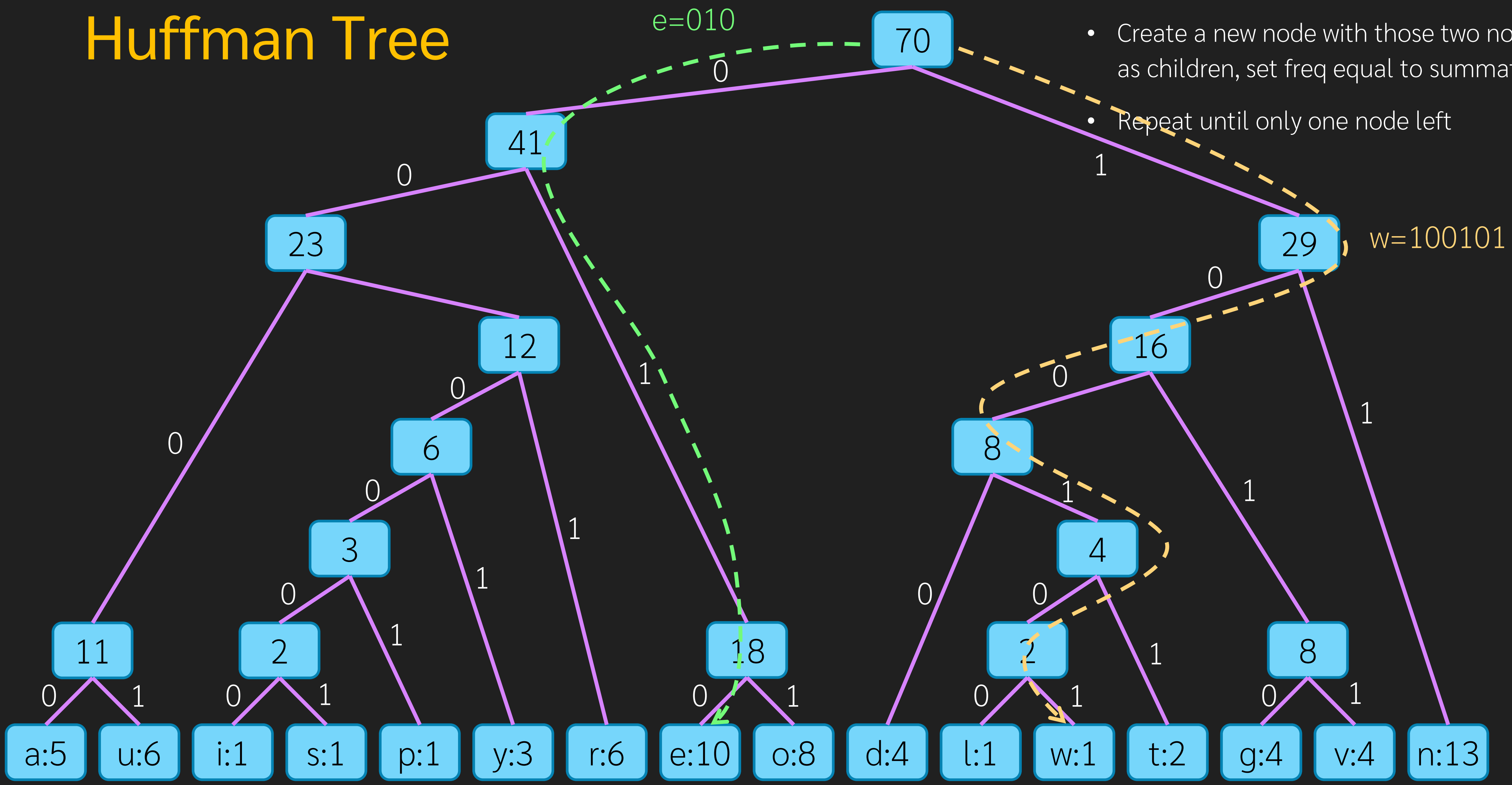
a = 010

b = 011

c = 1

Huffman Tree

- Find 2 min nodes
- Create a new node with those two nodes as children, set freq equal to summation
- Repeat until only one node left



Huffman Tree Node

- Instead of data, we have both character and frequency
- Since we have to pick two nodes with minimum freq, we overload operator< to do so and use priority_queue

Huffman Code : Node

```
class huffman_tree {
  protected:
    class huffman_node {
      public:
        char letter;
        int freq;
        huffman_node *left, *right;
        huffman_node() : letter('*'),freq(0),left(NULL),right(NULL) {}
        huffman_node(char letter,int freq,huffman_node *left,huffman_node *right) :
          letter(letter),freq(freq), left(left),right(right) {}

        bool is_leaf() { return left == NULL && right == NULL;  }
    };

    class node_comparator {
      public:
        bool operator()(const huffman_node *a, const huffman_node *b) {
          return a->freq > b->freq;
        }
    };
```

Huffman Code : Build Tree

```
class huffman_tree {
  protected:
    huffman_node *root;
    void build_tree(vector<huffman_node*> data) {
      priority_queue<huffman_node*,vector<huffman_node*>,node_comparator> pq;
      for (auto &x : data) pq.push(x);
      while (pq.size() > 1) {
        huffman_node *right = pq.top(); pq.pop();
        huffman_node *left = pq.top(); pq.pop();
        pq.push(new huffman_node('*',left->freq+right->freq,left,right));
      }
      root = pq.top();
    }
  public:
    huffman_tree(string s) {
      map<char,int> count;
      for (auto &c : s)
        count[c]++;
      vector<huffman_node*> nodes;
      for (auto &x : count)
        nodes.push_back(new huffman_node(x.first,x.second,NULL,NULL));
      build_tree(nodes);
    }
```

Recursive Programming

Calling itself

Recursive

Terminating condition

- A function that call itself
- Must have some input, usually via function argument
- The function must check a condition for execution
  - Result in either terminating case where the function won't call itself
  - or recursion case where the function will call itself with different parameters

```
// calculate sum 0..n
int recur1(int n) {
  if (n <= 0) {
    // terminating case
    return 0;
  } else {
    // recursion case
    return recur1(n-1) + n;
  }
}
```

Smaller parameter

Why recursion?

- Much simpler code
  - When the task is right
  - Recursion is natural for several mathematical model that is recursi
- Comparing to a normal loop, recursion has the same growth rate but recursion might takes more time because function call is costlier than a loop

More Example

```
void print_range1(int step,int goal) {
  if (step < goal) {
    std::cout << step << " ";
    print_range1(step+1, goal);
  }
}
```

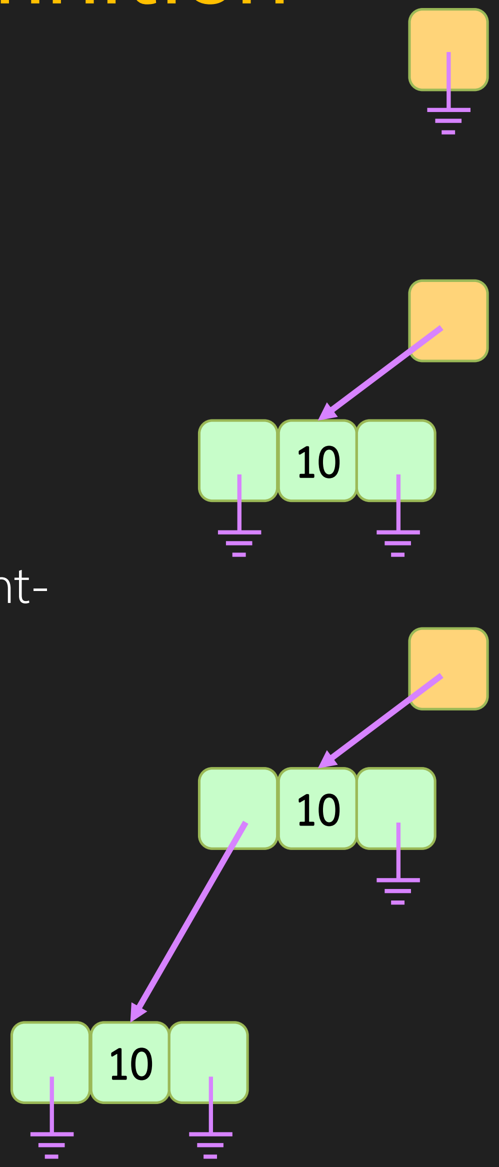
```
void print_range2(int step,int goal) {
  if (step < goal) {
    print_range2(step+1, goal);
    std::cout << step << " ";
  }
}
```

- Terminating Case do nothing
- Which is the output of print_range1(0,5) and print_range2(0,5)

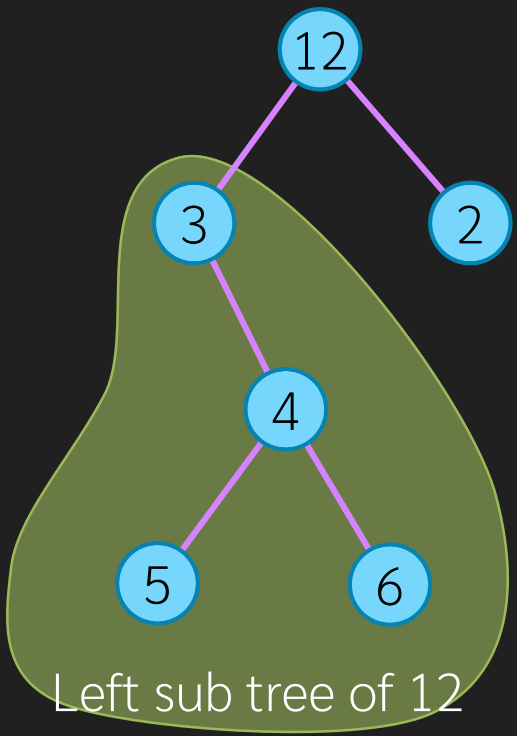
0 1 2 3 4 5	0 1 2 3 4
5 4 3 2 1 0	4 3 2 1 0

Binary Tree Recursive Definition

- A Binary Tree is
  - A tree with no nodes (root is NULL)
  - A tree with a root
    - both children of the root must be a binary tree
    - Each child is call left-subtree and right-subtree
- Since binary tree can be defined recursively, operation on a binary tree can be naturally written as a recursion



Subtree

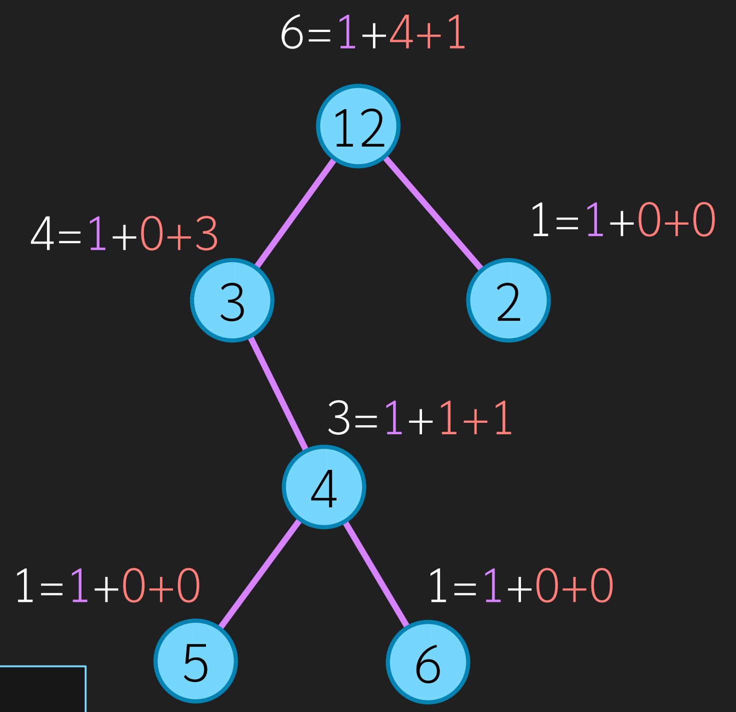


- For any node
  - its left (right) child and all of the child's descendants is called left-subtree (right-subtree)

Tree Size by Recursion

- An empty tree has 0 node
- A tree with a root has 1 node (the root)
  - Plus the size of its two subtrees
- Easily written as recursive

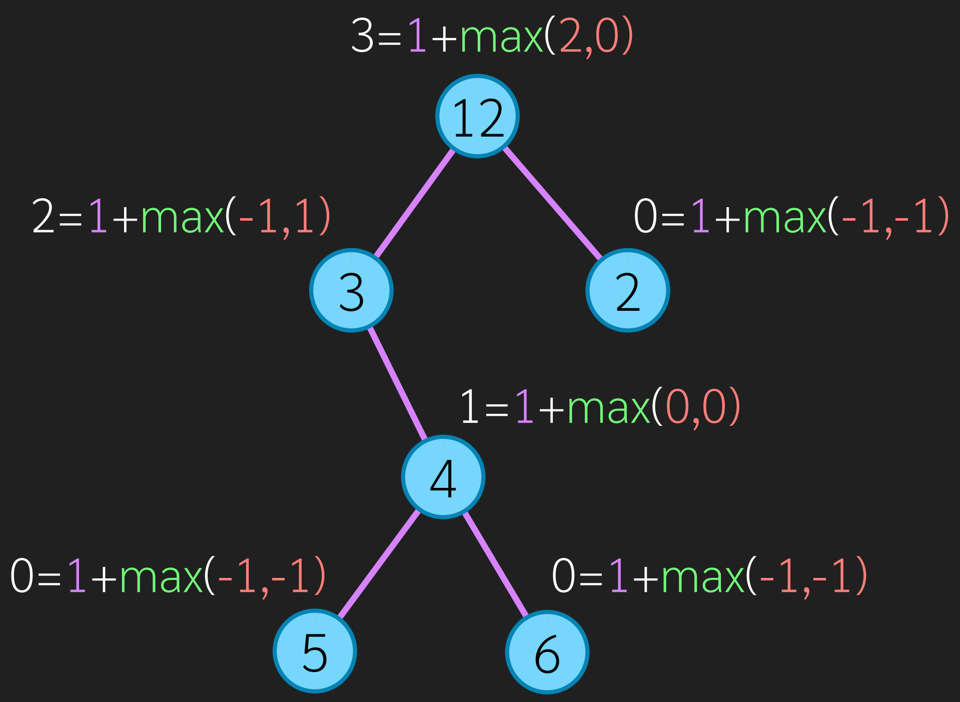
```
class node {
  public:
    int data;
    node *left, *right;
};

int get_size(node* n) {
  if (n == NULL) return 0;
  return 1 + get_size(n->left) + get_size(n->right);
}
```



Tree Height

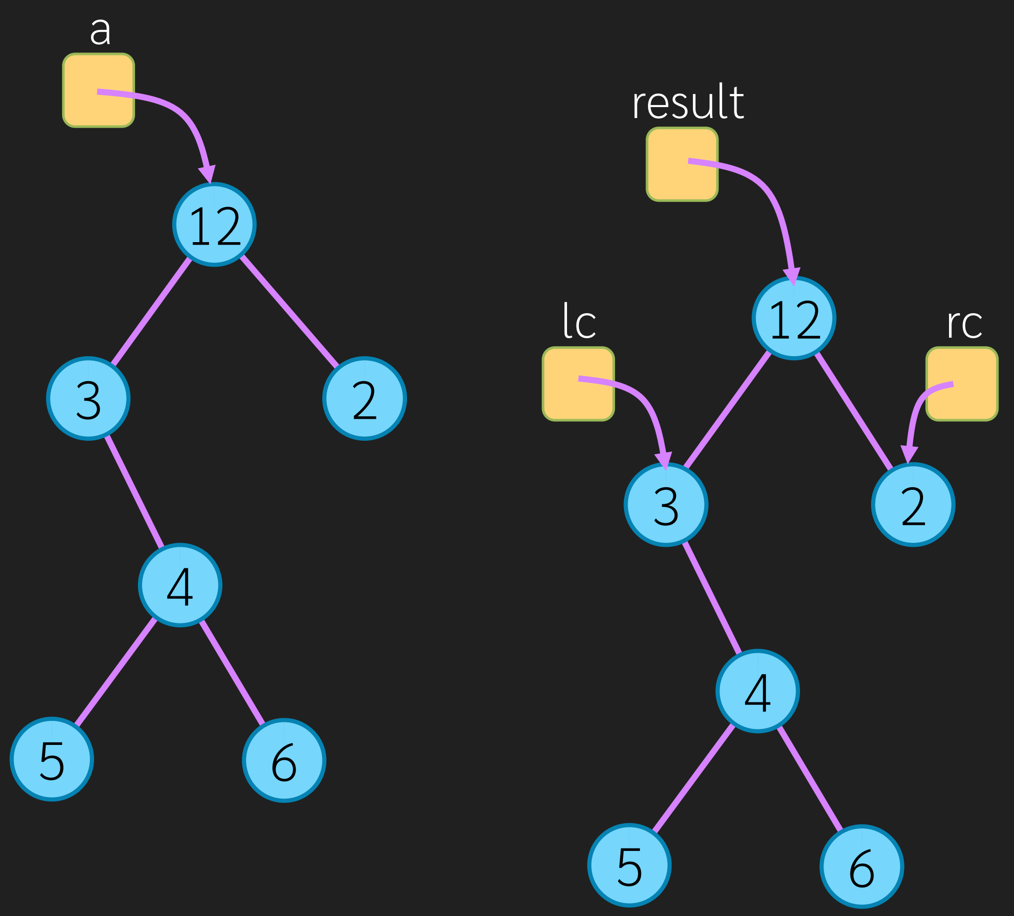
- Height of a tree is the number of link we have to go to reach it deepest children
- Empty tree has height -1
- Height of a tree is 1 + max of height of its children

```
class node {
  public:
    int data;
    node *left, *right;
};
int get_height(node *n) {
  if (n == NULL) return -1;
  return 1 + std::max(get_height(n->left),
                      get_height(n->right));
}
```



Tree Copy

```
class node {
  public:
    int data;
    node *left, *right;
    node() : data(0),left(NULL),right(NULL);
    node(int data,node *left,node *right)
      : data(data),left(left),right(right);
};

node* copy(node *n) {
  if (n == NULL) return NULL;
  node *lc = copy(n->left);
  node *rc = copy(n->right);
  node *result = new node(n->data,lc,rc);
}
```



Walk over a tree

- Visiting all nodes (and maybe do something)

```
void preorder(node *n) {
  if (n == NULL) return NULL;
  std::cout << n->data << " ";
  preorder(n->left);
  preorder(n->right);
}
```

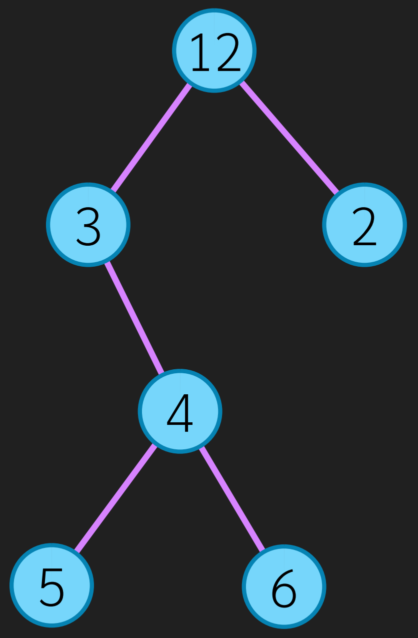
preorder traversal

```
void inorder(node *n) {
  if (n == NULL) return NULL;
  inorder(n->left);
  std::cout << n->data << " ";
  inorder(n->right);
}
```

inorder traversal

```
void postorder(node *n) {
  if (n == NULL) return NULL;
  postorder(n->left);
  postorder(n->right);
  std::cout << n->data << " ";
}
```

postorder traversal



What is the result of

- preorder(a);
- inorder(a);
- postorder(a);

Huffman Tree : Encoding

```
class huffman_tree {
  protected:
    class huffman_node { };
    class node_comparator {  };
    huffman_node *root;
  public:
    void print(huffman_node *n,string s) {
      if (n->is_leaf()) {
        cout << n->letter << ": " << s << endl;
      } else {
        print(n->left,s+"0");
        print(n->right,s+"1");
      }
    }

    void print() {
      print(root,"");
    }
};
```

- Recursive printing
- Use s to store path

Huffman Tree : Encoding

```
class huffman_tree {
  protected:
    class huffman_node { };
    class node_comparator {  };

    huffman_node *root;

    void delete_node(huffman_node *n) {
      if (n == NULL) return;
      delete_node(n->left);
      delete_node(n->right);
      delete n;
    }

  public:

    ~huffman_tree() {
      delete_node(root);
    }
};
```

- Recursive delete node
- Use postorder traversal
- Can we use inorder or preorder?

Binary Search Tree

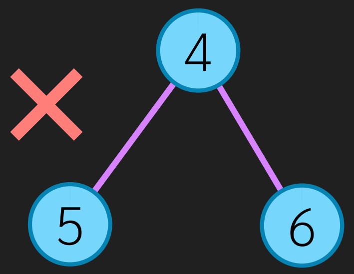
Binary Tree with value condition

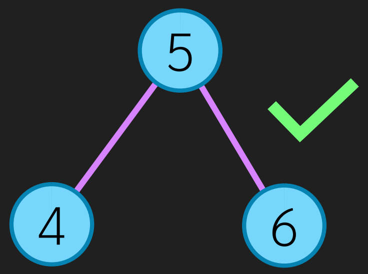
Overview

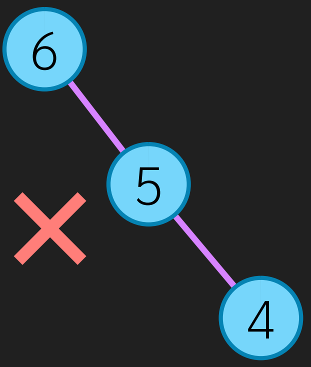
- We add additional value constraint to a Binary Tree
- The constraint make finding data in the tree much faster
  - $O(h)$ where h is the height of the tree
  - The tree is expected to have h be in $O(\lg n)$, but this is not always true
  - The next tree (AVL tree) will add more constraint so that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

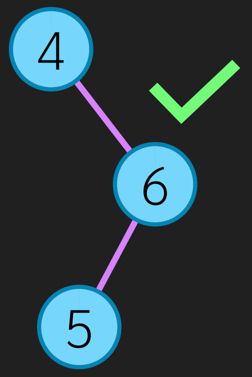
Binary Search Tree

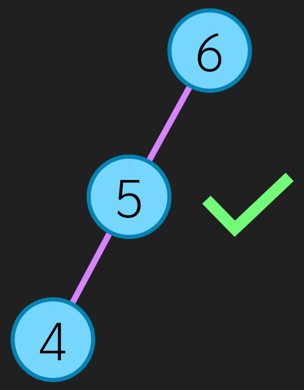
- Structure rule: must be a Binary Tree
- Value rule: for any node x
  - data in left-subtree must be less than the data in x
  - data in right-subtree must be more than the data in x
- Recursive Definition
  - An empty tree is a Binary Search Tree (BST)
  - A node X is a BST when
    - Its subtrees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





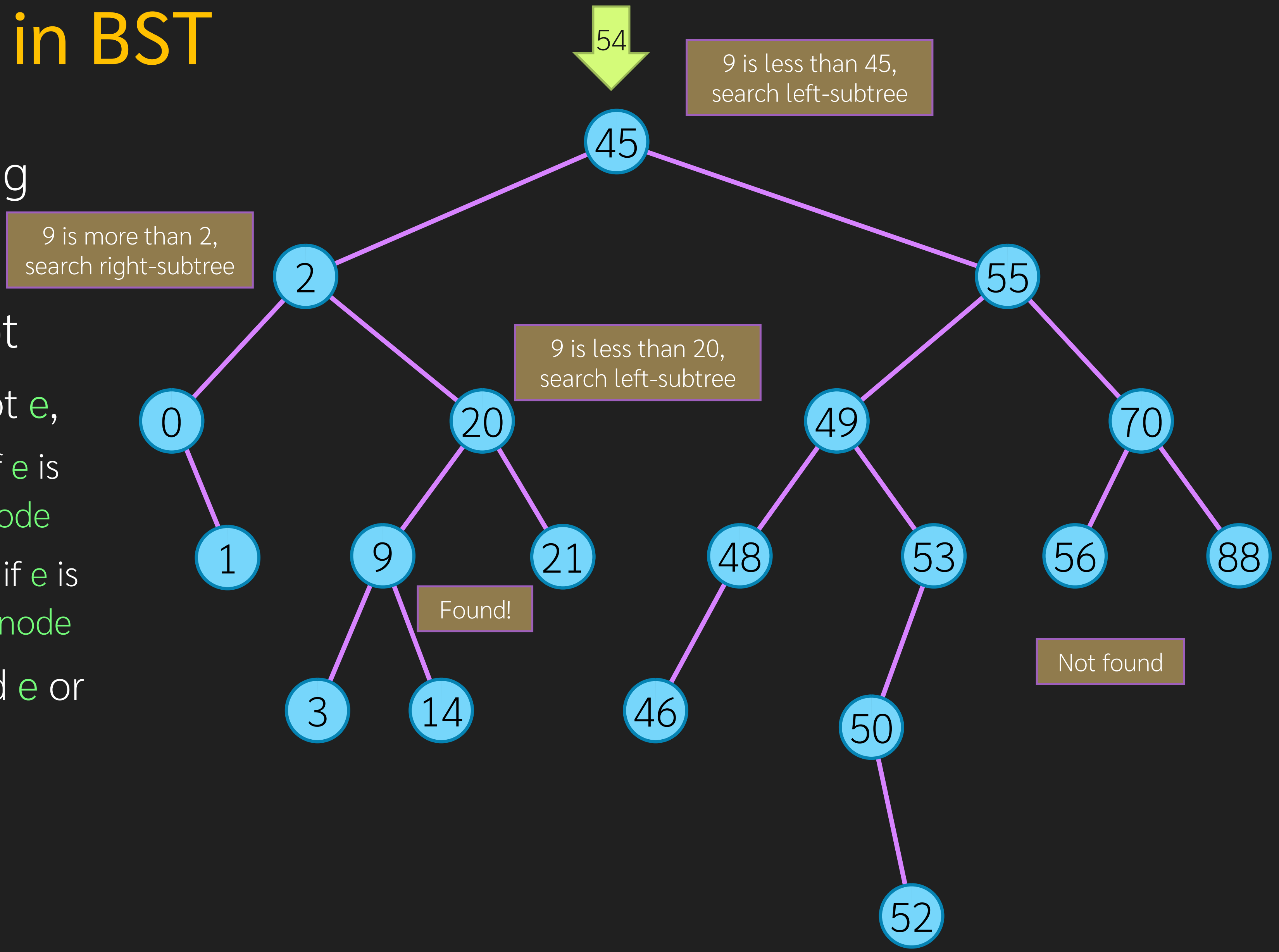






Finding Value in BST

- 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

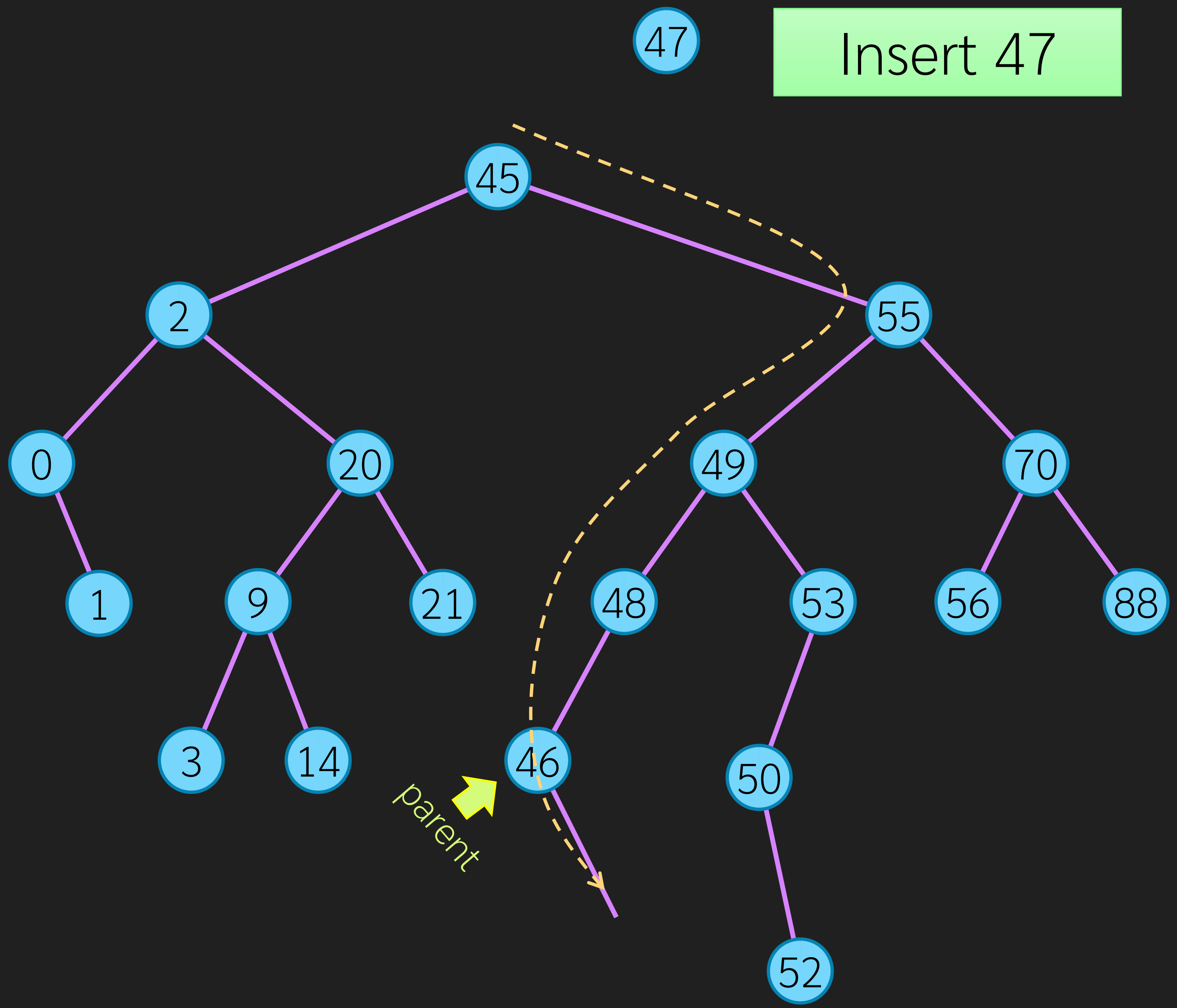
Later, we will need a parent node of the searching value

```
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 ) { }
};
```

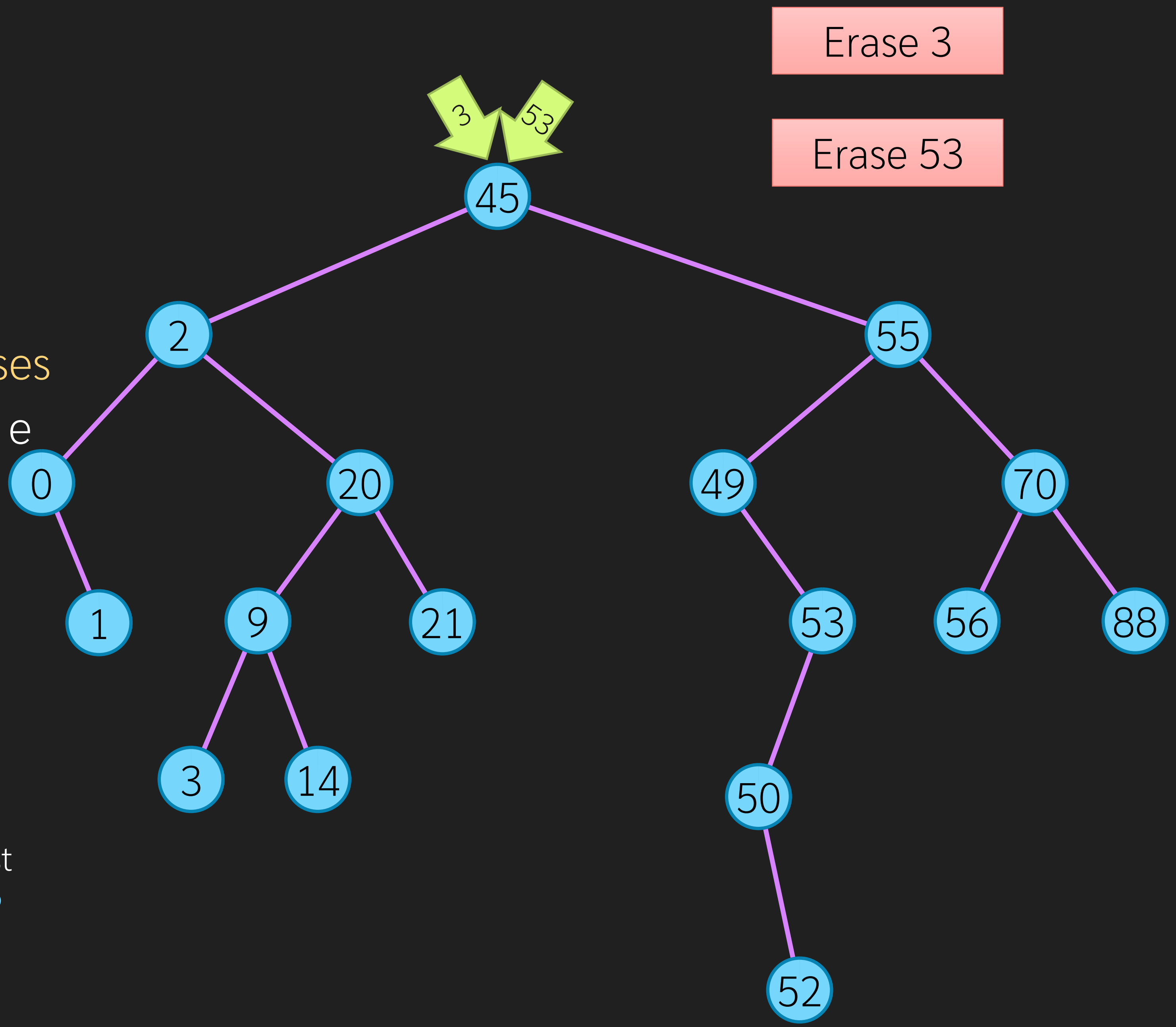
Insert

- 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



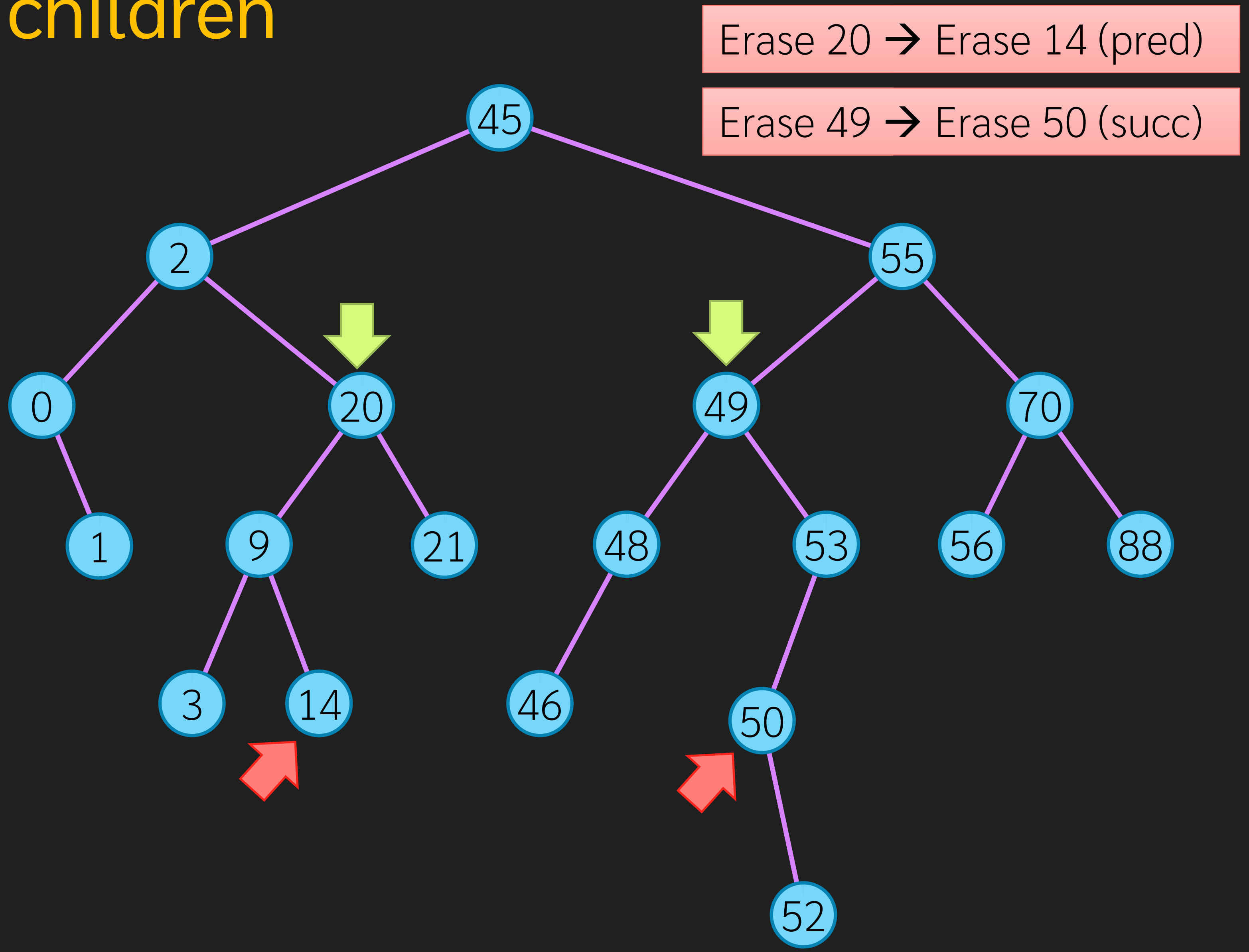
Erase

- 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

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

- 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
  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

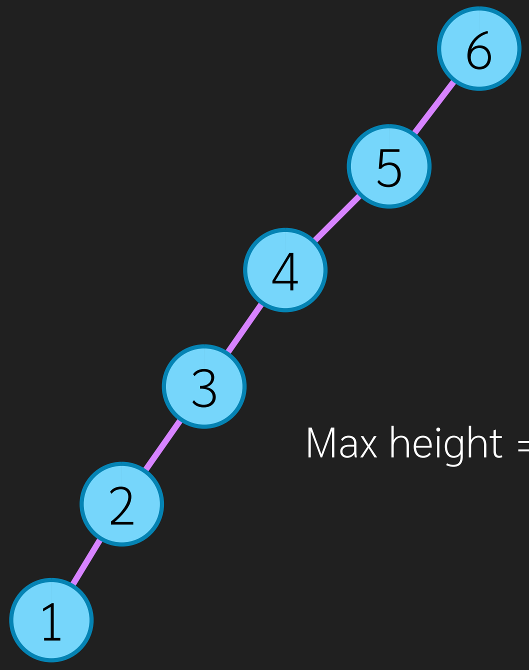
```
node* find_min_node(node* r) {
  //r must not be NULL
  if (r->left == NULL) return r;
  return find_min_node(r->left);
}
```

- 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

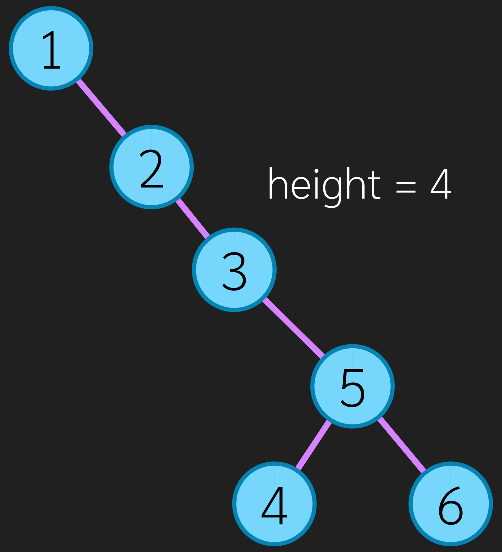
```
node* find_max_node(node* r) {
  //r must not be NULL
  if (r->right == NULL) return r;
  return find_max_node(r->right);
}
```

Complexity Analysis

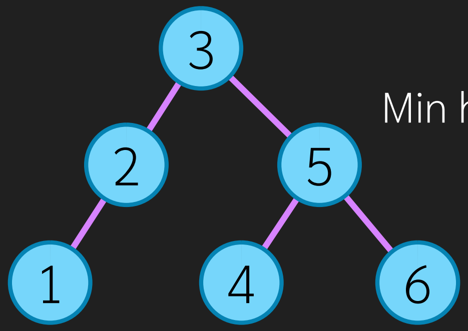
- Insert, erase depends in find, find_min (or find_max)
- All finds start from root and in the worst case reach the leaf
  - Hence, O(h)
- Height of the tree can be in the range from n to lg n
- For 1,000,000 nodes, its in the range of [20,999999]
  - O(h) is, right now, O(n)
  - Will be fixed by AVL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



Max height = 5



height = 4



Min height = 2

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

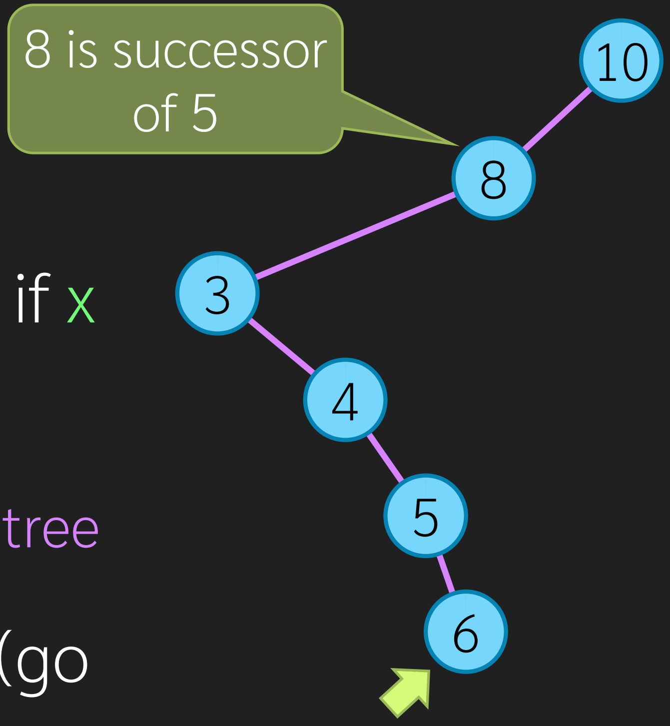
  MappedTypeReturn MappedType of the node

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
};
```

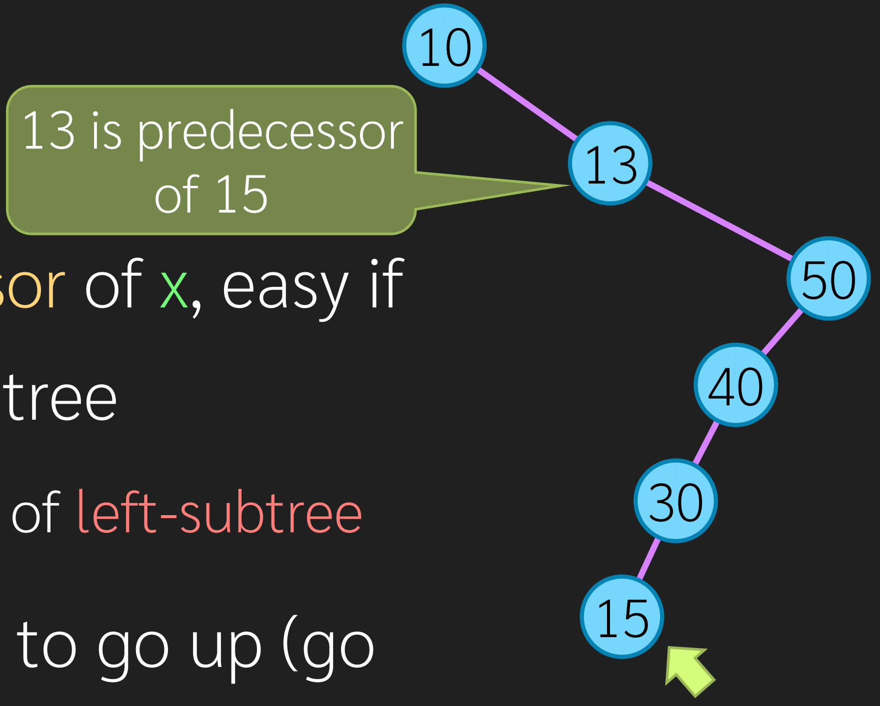
Operator++



- 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)