Binary Tree

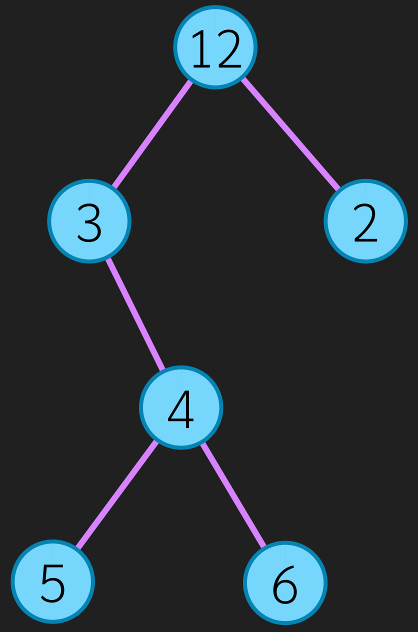
Practicing Pointer & Recursive

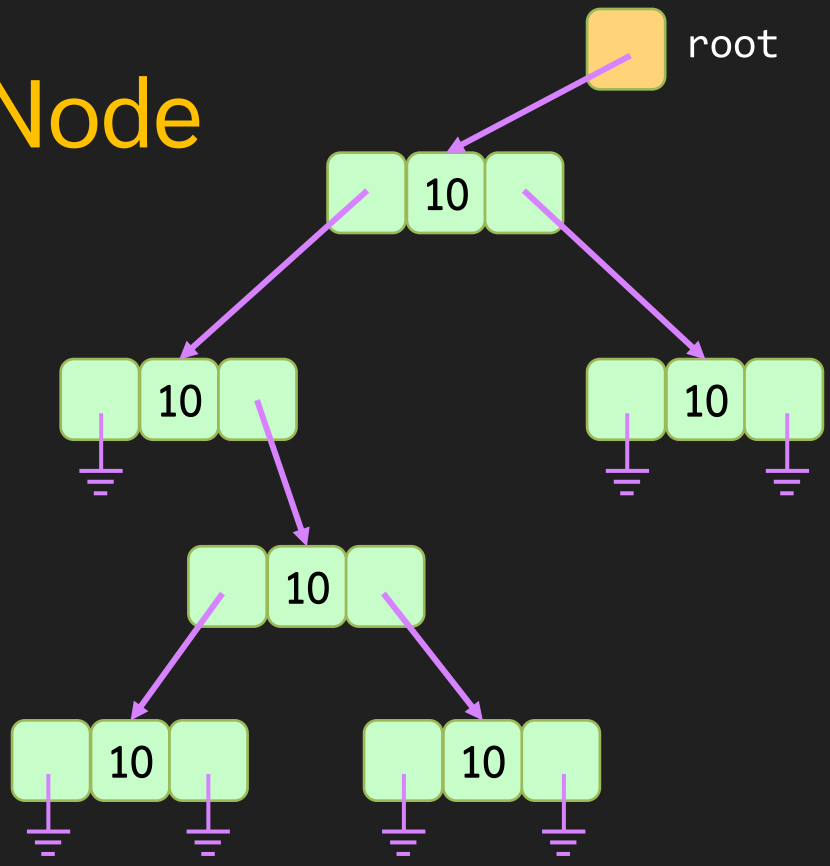
Nattee Niparnan

Overview

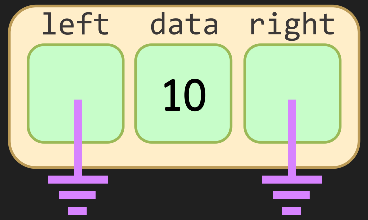
- This is a basic for the next data structure, Binary Search and AVL Tree
- Focus on using Node and Pointer
- Focus on using recursive programming
- Some applications using just Binary Tree
- There is no data structure in std that is Binary Tree

Binary Tree & Node



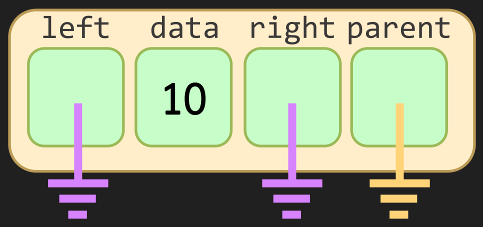


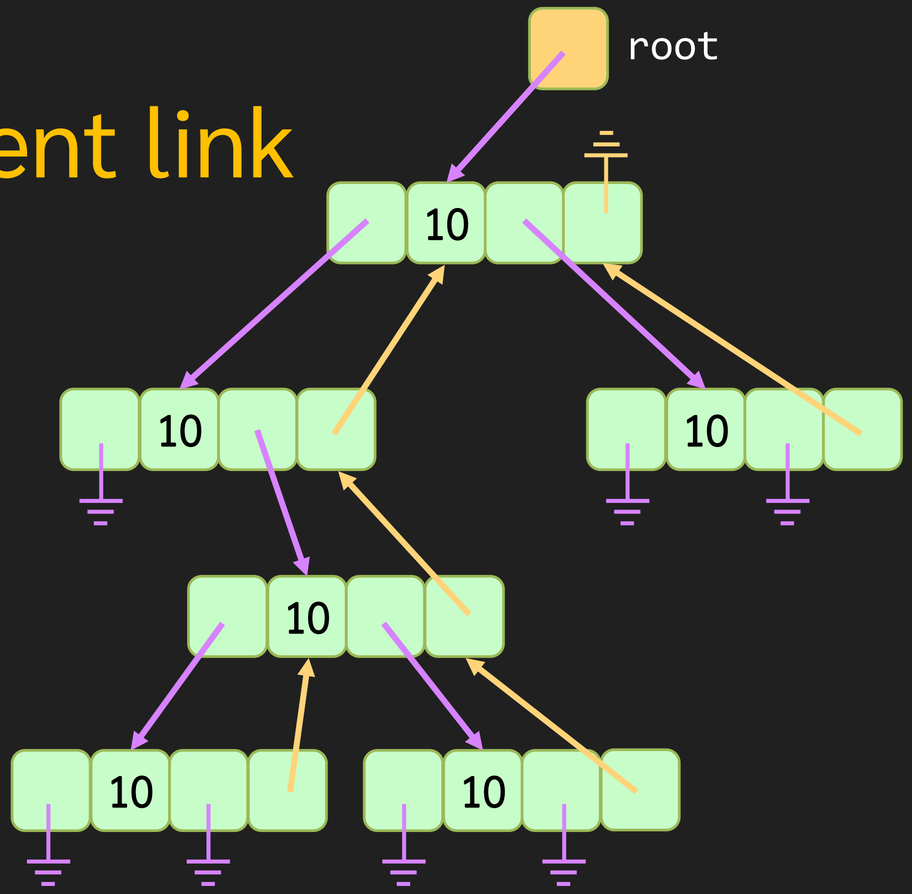
- A rooted tree where each node have at most two children
- Tree Node is very similar to a linked list node




```
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	0010 1	1001 1	1001 01	0010 001	0010 01	1001 00	0010 000

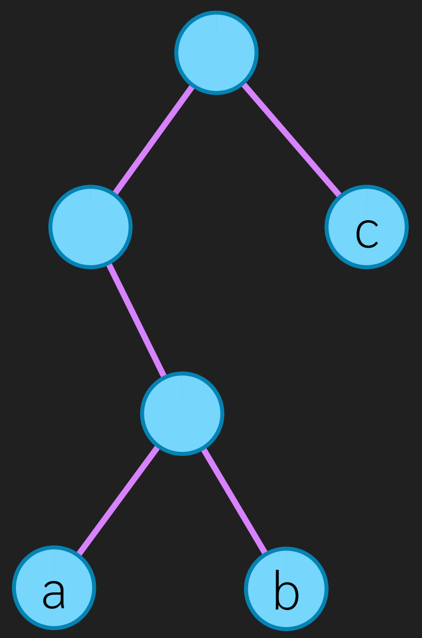
Encoding “Never”
Fix-length 00000001011000010100
Variable Legnth 1101010110100011

Problem Statement

- Input: a string
- Output: encoding of each character in the string such that
  - The total length of encoding the string is minimum
  - 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 represented 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



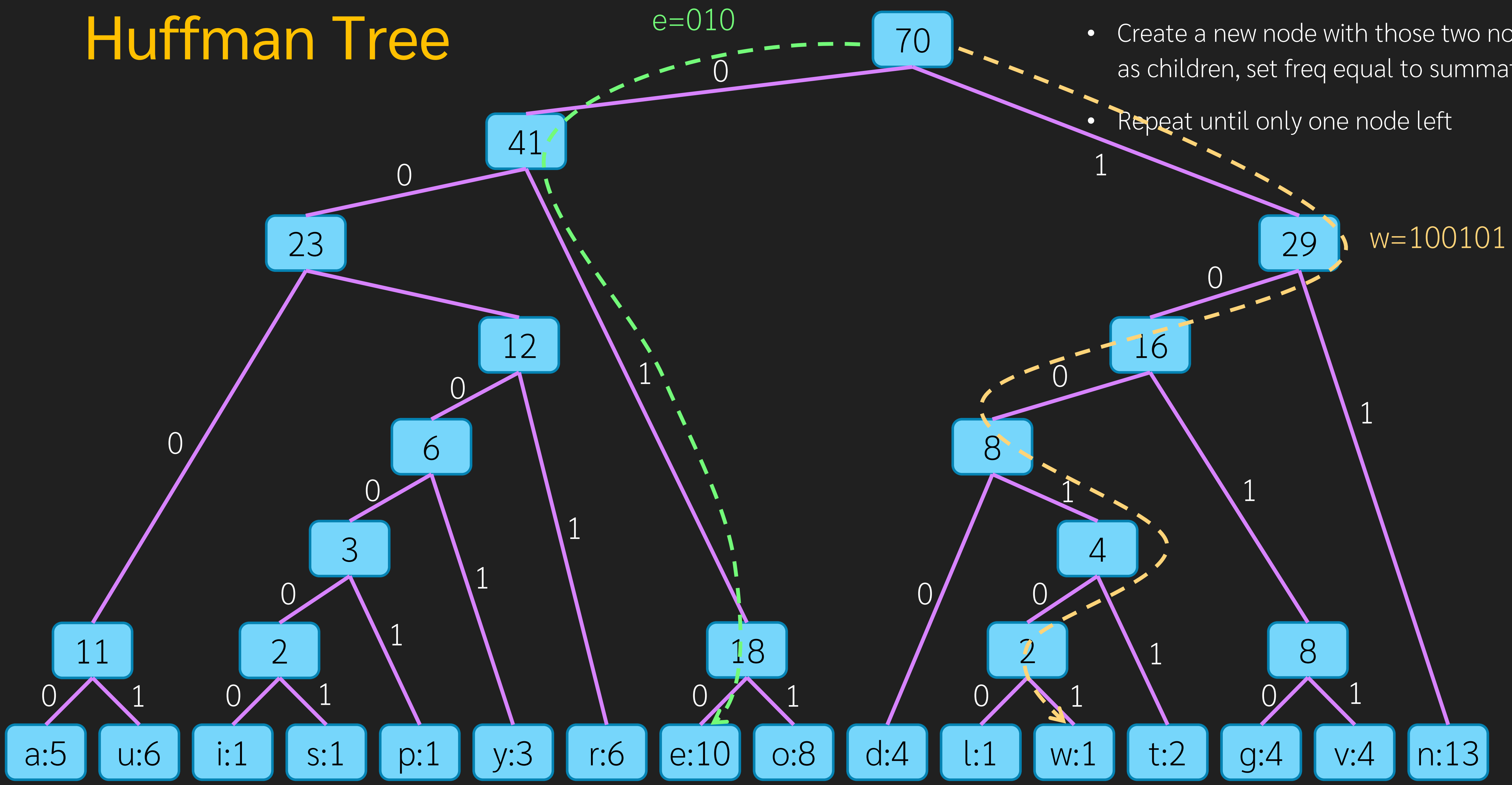
a = 010

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);
  }
}
```

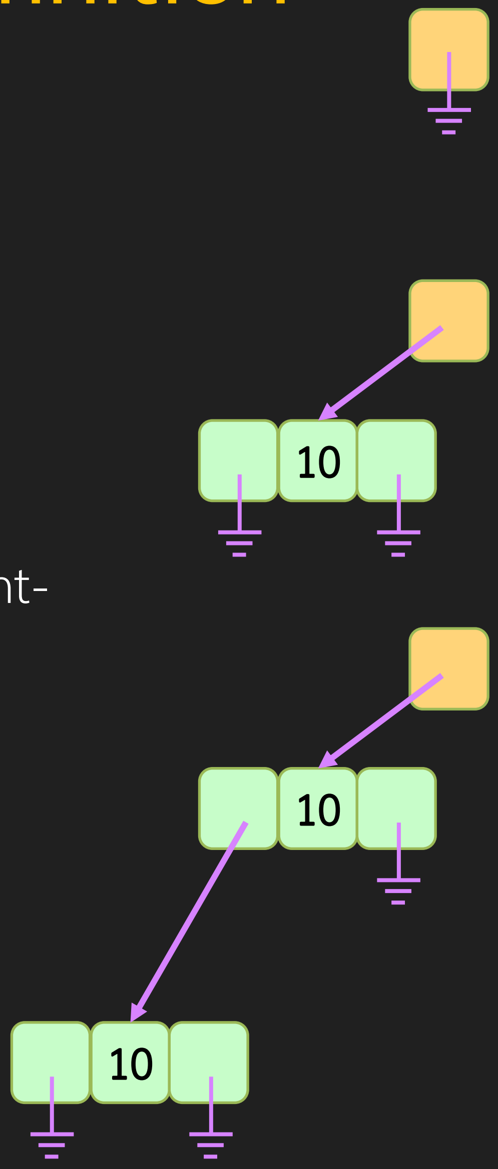
```
void print_range2(int step,int goal) {
  if (step < goal) {
    print_range2(step+1, goal);
    std::cout << step << " ";
  }
}
```

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

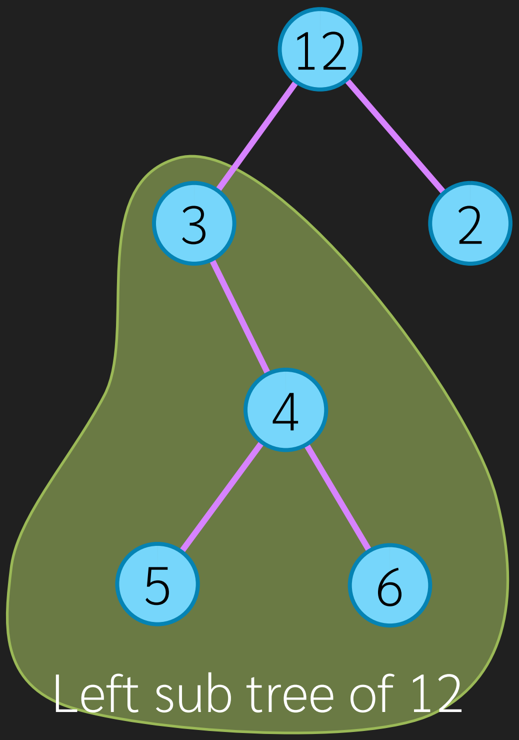
0 1 2 3 4 5	0 1 2 3 4
5 4 3 2 1 0	4 3 2 1 0

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

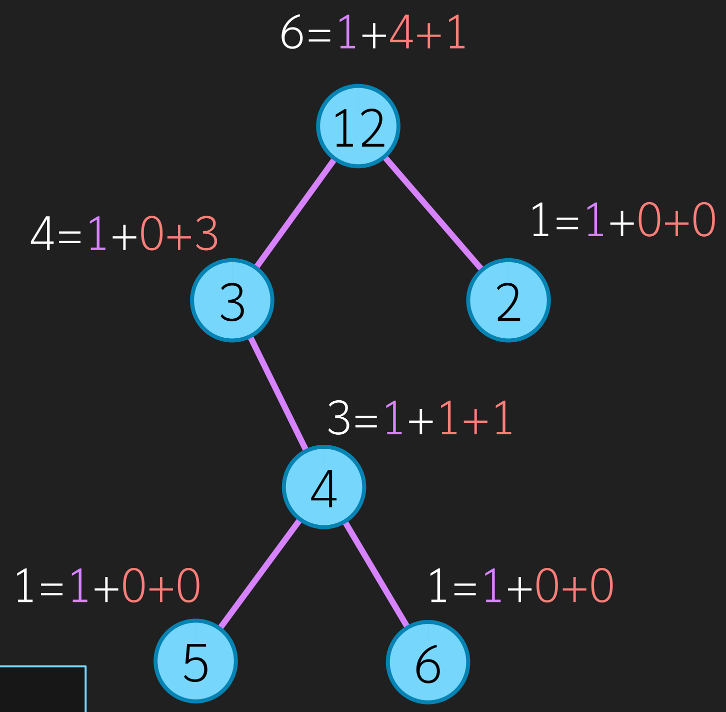


- For any node
  - its left (right) child and all of the child's descendants is called left-subtree (right-subtree)

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

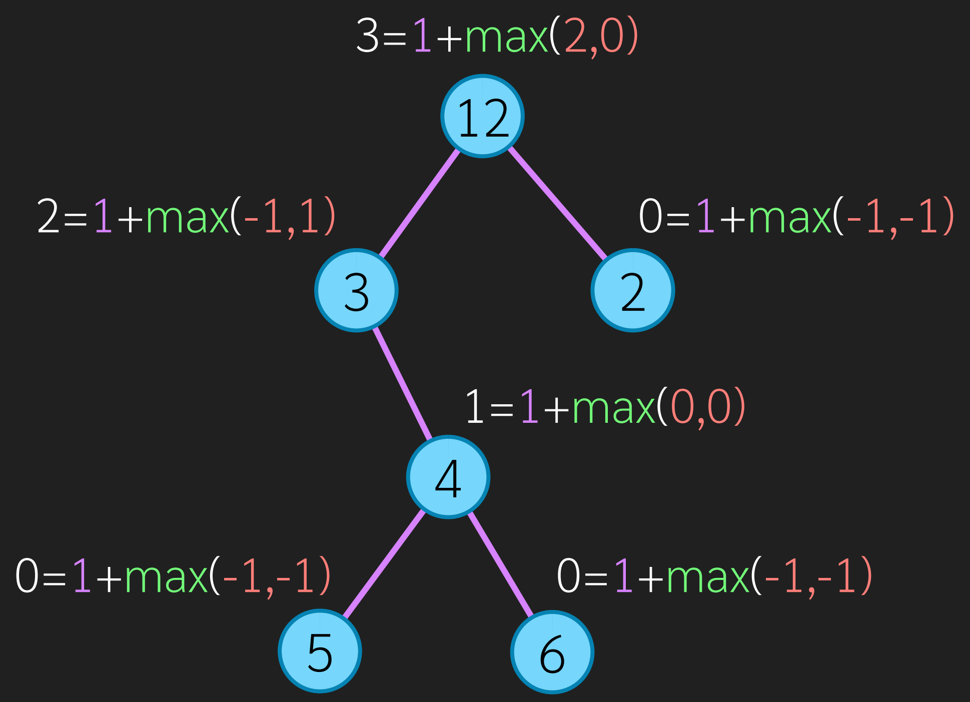
```
class node {
  public:
    int data;
    node *left, *right;
};

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



Tree Height

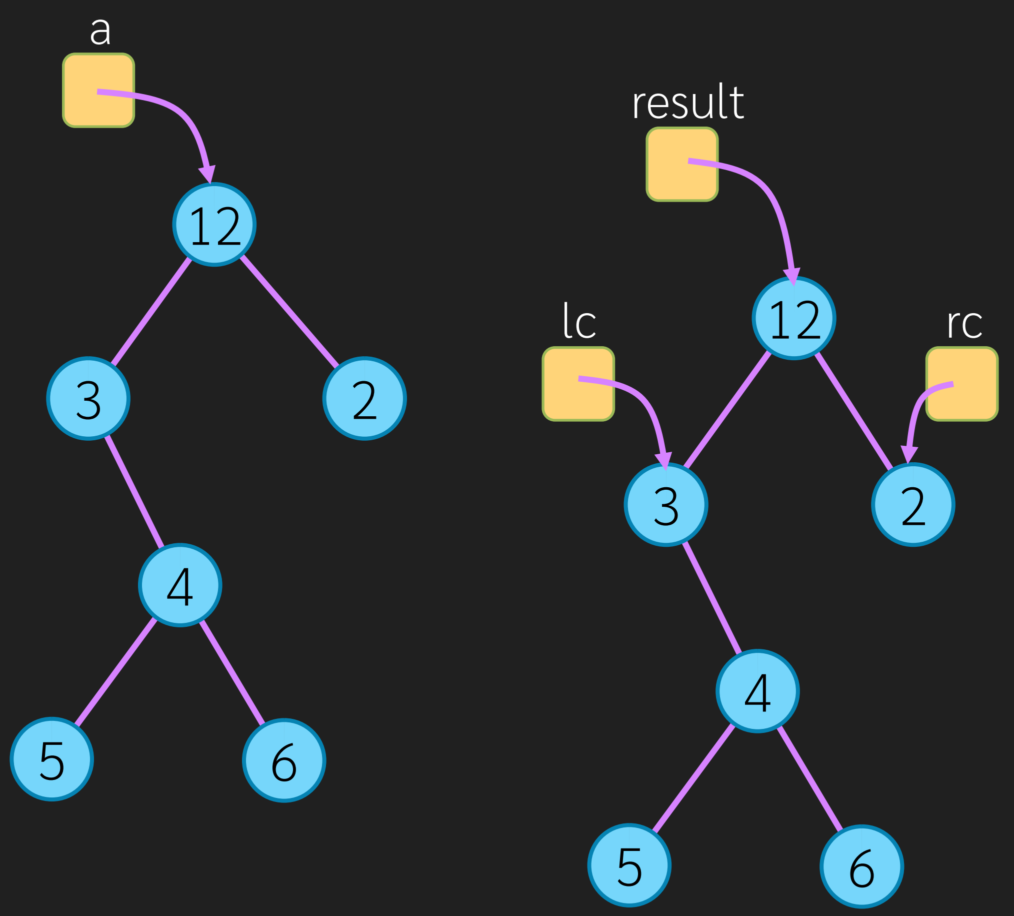
- Height of a tree is the number of link we have to go to reach it deepest children
- Empty tree has height -1
- Height of a tree is 1 + max of height of its children

```
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);
}
```

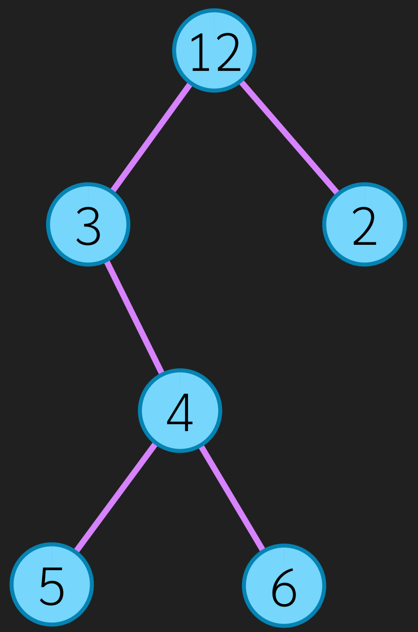
preorder traversal

```
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?