CS 225

Data Structures

September 24 – Iterators and Trees

G Carl Evans

stlList.cpp

```
#include <list>
#include <string>
#include <iostream>

struct Animal {
  std::string name, food;
  bool big;
  Animal(std::string name = "blob", std::string food = "you", bool big = true) :
    name(name), food(food), big(big) { /* nothing */ }
};

int main() {
  Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
  std::vector<Animal> zoo;

  zoo.push_back(g);
  zoo.push_back(p);   // std::vector’s insertAtEnd
  zoo.push_back(b);

  for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); it++ ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }

  return 0;
}
```

stlList.cpp

```
#include <list>
#include <string>
#include <iostream>

struct Animal {
  std::string name, food;
  bool big;
  Animal(std::string name = "blob", std::string food = "you", bool big = true) :
    name(name), food(food), big(big) { /* nothing */ }
};

int main() {
  Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
  std::vector<Animal> zoo;

  zoo.push_back(g);
  zoo.push_back(p);   // std::vector's insertAtEnd
  zoo.push_back(b);

  for ( auto it = zoo.begin(); it != zoo.end(); it++ ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }

  return 0;
}
```

stlList.cpp

```
#include <list>
#include <string>
#include <iostream>

struct Animal {
  std::string name, food;
  bool big;
  Animal(std::string name = "blob", std::string food = "you", bool big = true) :
    name(name), food(food), big(big) { /* none */ }
};

int main() {
  Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
  std::vector<Animal> zoo;

  zoo.push_back(g);
  zoo.push_back(p);   // std::vector's insertAtEnd
  zoo.push_back(b);

  for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
  }

  return 0;
}
```

For Each and Iterators

```
for ( const TYPE & variable : collection ) {
  // ...
}
```

```
std::vector<Animal> zoo;
…
for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
}
```

For Each and Iterators

```
for ( const TYPE & variable : collection ) {
  // ...
}
```

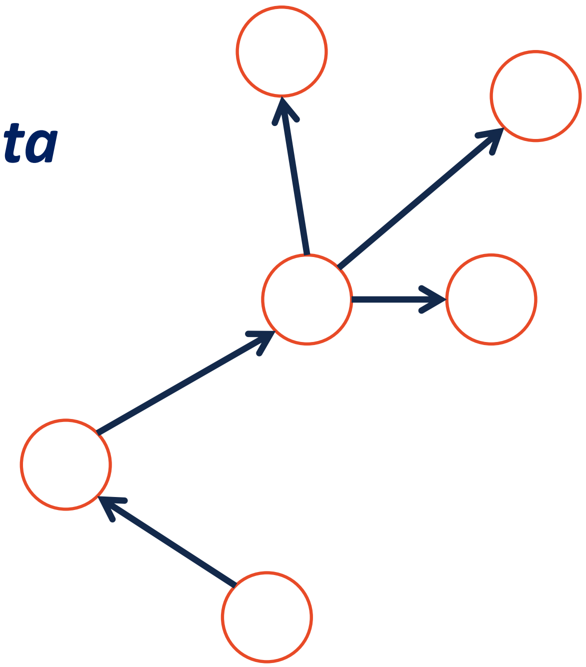
```
std::vector<Animal> zoo;
…
for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
}
```

```
std::unordered_set<std::string, Animal> zoo;
…
for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
}
```

Trees

"The most important non-linear data structure in computer science."

- David Knuth, The Art of Programming, Vol. 1



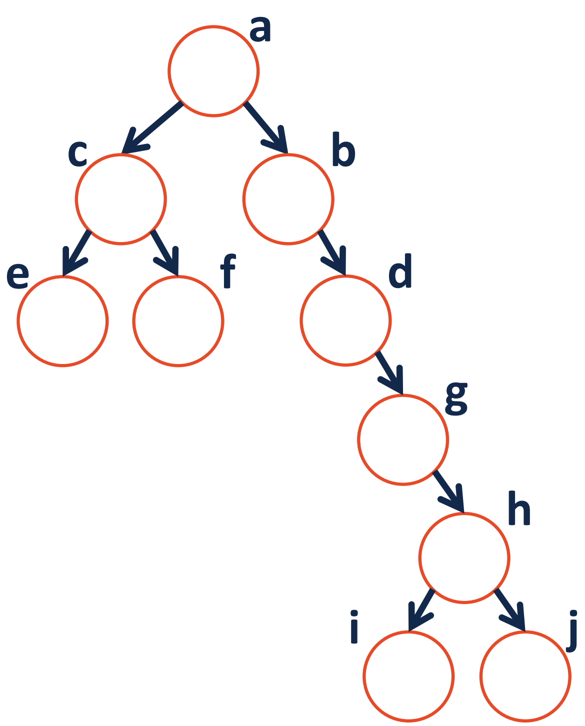
A tree is:

-
-

More Specific Trees

We'll focus on **binary trees**:

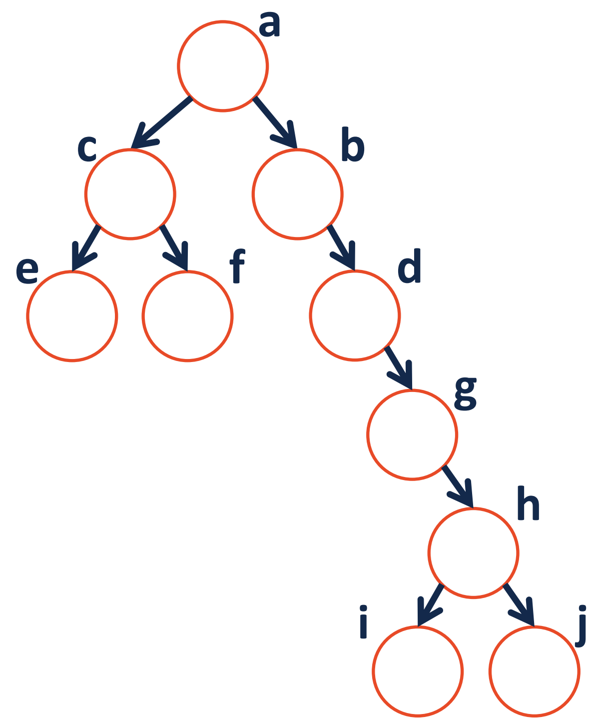
- A binary tree is **acyclic** – there are no cycles within the graph



More Specific Trees

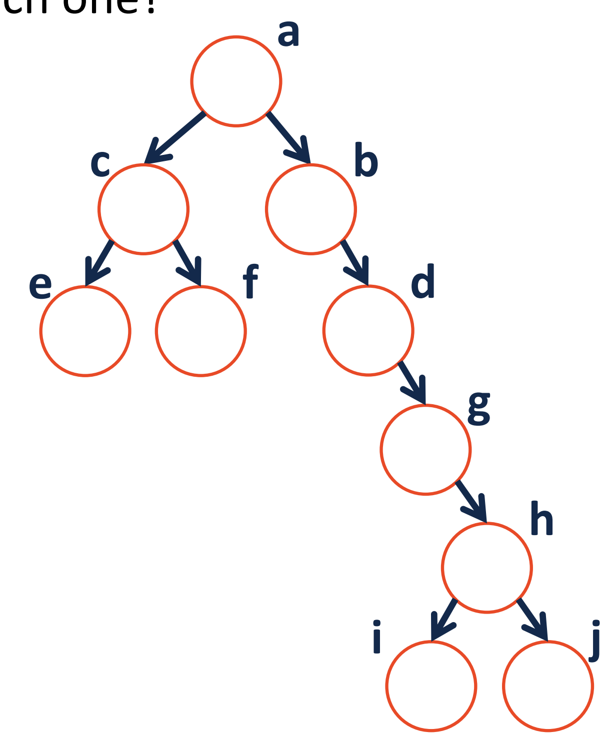
We'll focus on **binary trees**:

- A binary tree contains **two or fewer children** – where one is the "left child" and one is the "right child":



Tree Terminology

- Find an **edge** that is not on the longest **path** in the tree. Give that edge a reasonable name.
- One of the vertices is called the **root** of the tree. Which one?
- How many parents does each vertex have?
- Which vertex has the fewest **children**?
- Which vertex has the most **ancestors**?
- Which vertex has the most **descendants**?
- List all the vertices is b’s left **subtree**.
- List all the **leaves** in the tree.



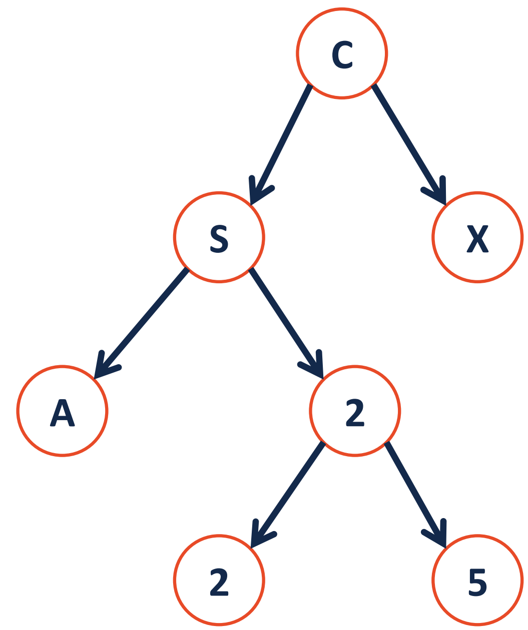
Binary Tree – Defined

A *binary tree* T is either:

-

OR

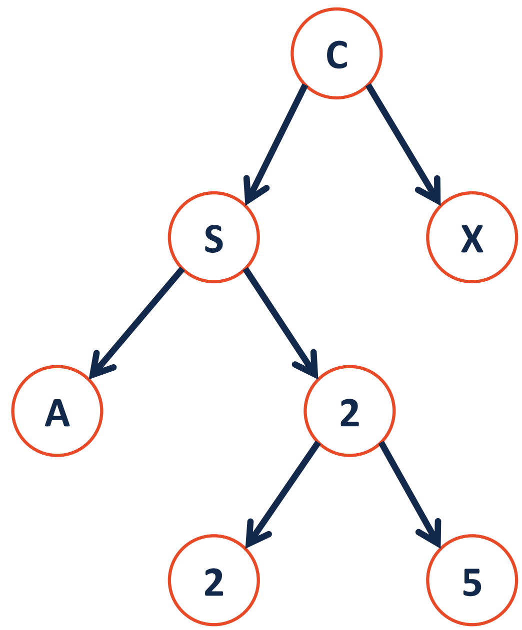
-



Tree Property: height

height(T)**:** length of the longest path from the root to a leaf

Given a binary tree T:



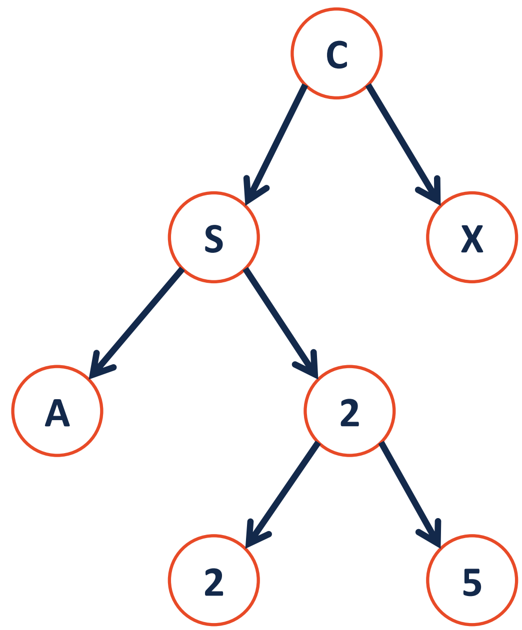
height(T) =

Tree Property: full

A tree ***F*** is **full** if and only if:

1.

2.

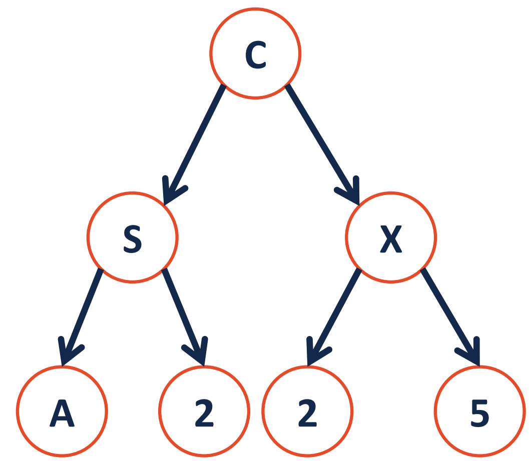


Tree Property: perfect

A **perfect** tree ***P*** is:

1.

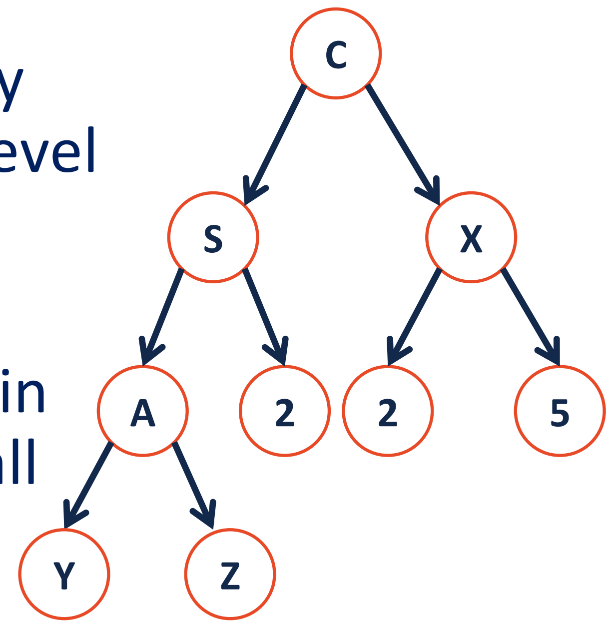
2.



Tree Property: complete

Conceptually: A perfect tree for every level except the last, where the last level if "pushed to the left".

Slightly more formal: For any level k in [0, h-1], k has 2^k nodes. For level h, all nodes are "pushed to the left".



Tree Property: complete

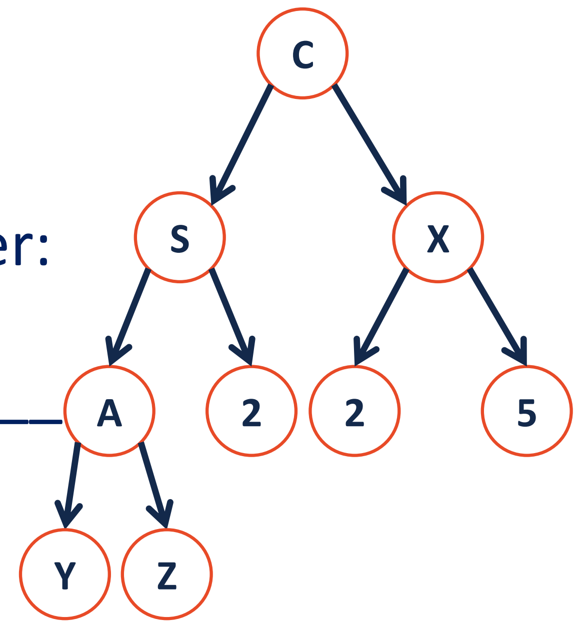
A **complete** tree ***C*** of height **h**, $\mathbf{C_h}$:

1. $\mathbf{C_{-1}}$ = {}
2. $\mathbf{C_h}$ *(where h>0)* **= {r, T_L, T_R}** and either:

T_L is ___________ and T_R is ___________

OR

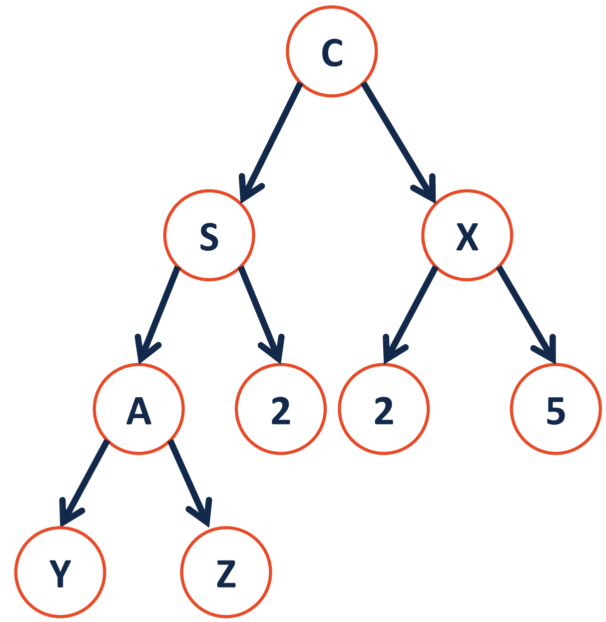
T_L is ___________ and T_R is ___________



Tree Property: complete

Is every **full** tree **complete**?

If every **complete** tree **full**?



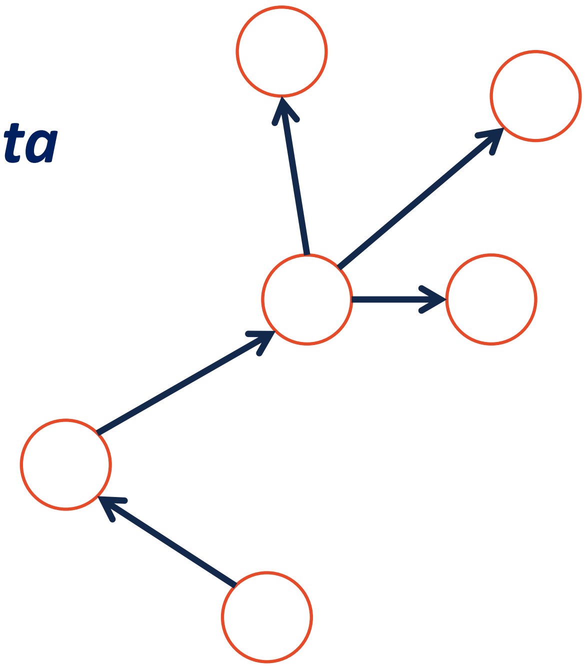
Trees

“The most important non-linear data structure in computer science.”

- David Knuth, The Art of Programming, Vol. 1

A tree is:

-
-



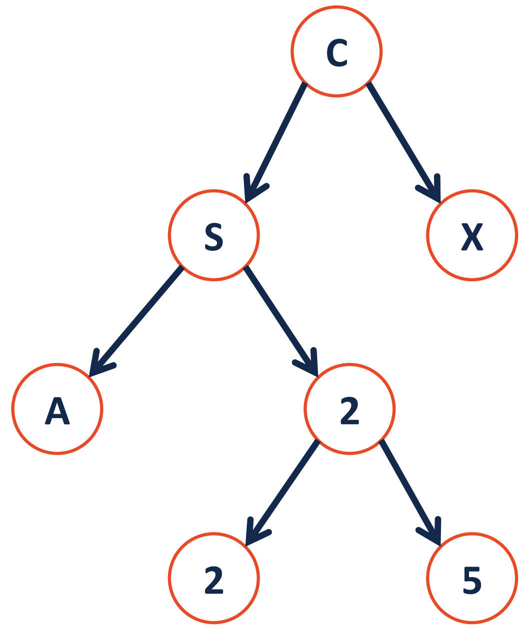
Binary Tree – Defined

A *binary tree* T is either:

-

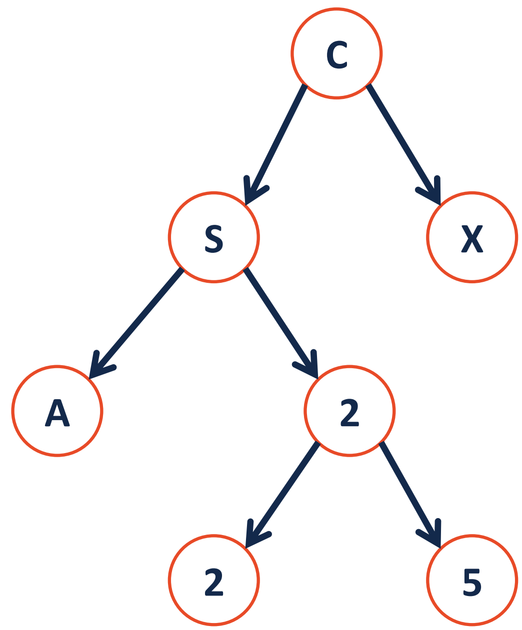
OR

-



Tree Property: height

height(T): length of the longest path from the root to a leaf



Given a binary tree T:

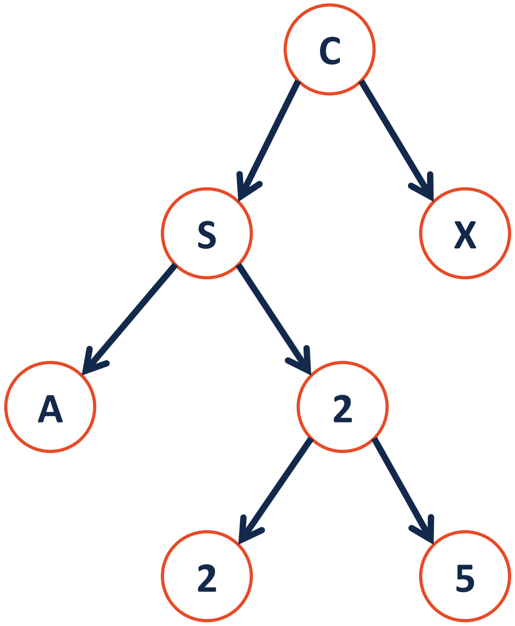
height(T) =

Tree Property: full

A tree ***F*** is **full** if and only if:

1.

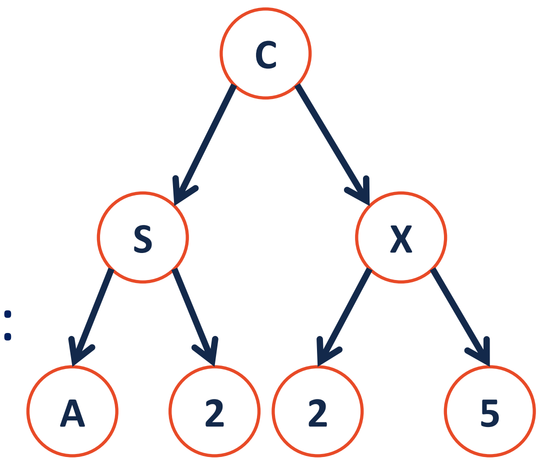
2.



Tree Property: perfect

A **perfect** tree ***P*** is defined in terms of the tree's height.

Let $\mathbf{P_h}$ be a perfect tree of height **h**, and:



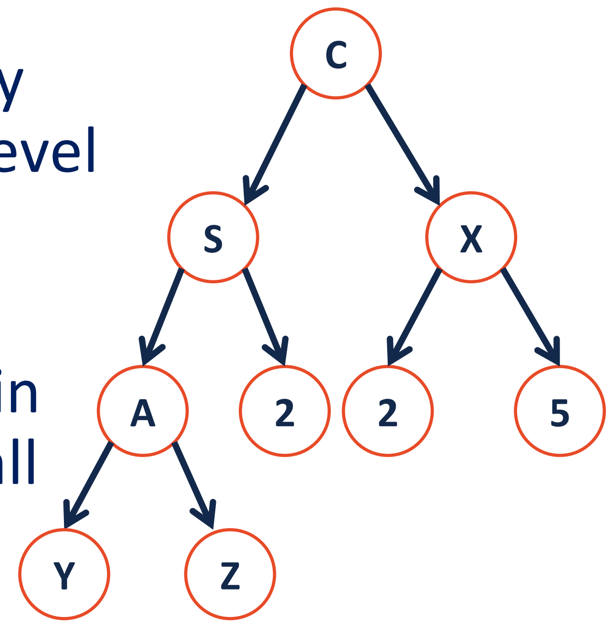
1.

2.

Tree Property: complete

Conceptually: A perfect tree for every level except the last, where the last level if "pushed to the left".

Slightly more formal: For all levels k in [0, h-1], k has 2^k nodes. For level h, all nodes are "pushed to the left".



Tree Property: complete

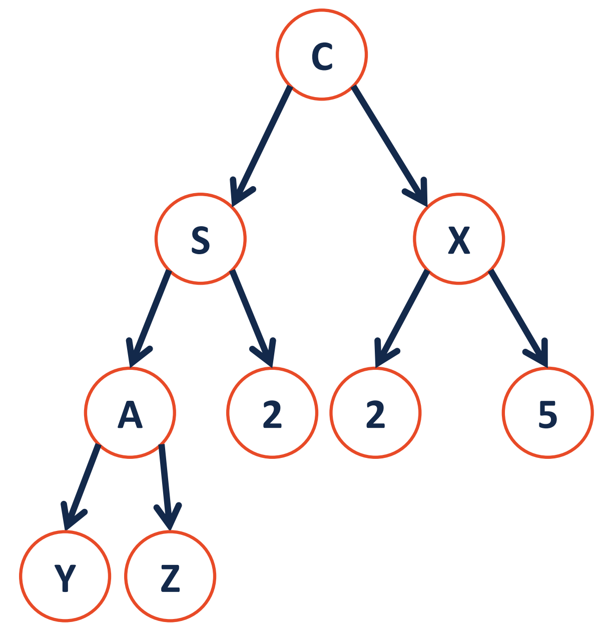
A **complete** tree ***C*** of height **h**, $\mathbf{C_h}$:

1. $\mathbf{C_{-1}}$ = {}
2. $\mathbf{C_h}$ *(where h>0)* **= {r, T_L, T_R}** and either:

T_L is ___________ and T_R is __________

OR

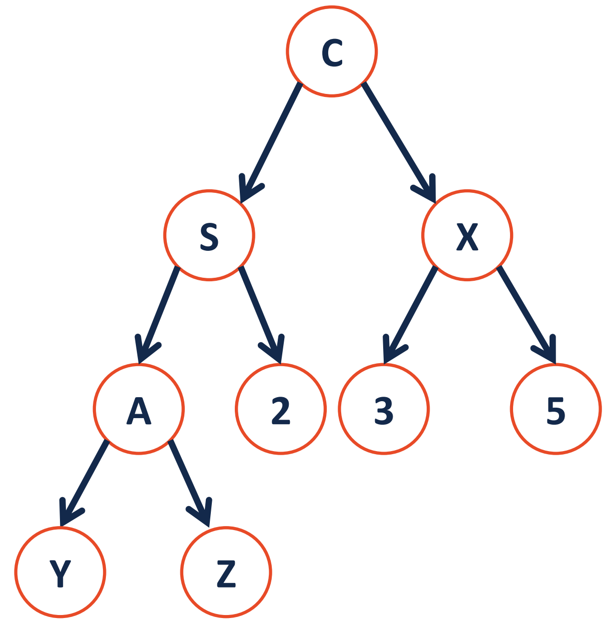
T_L is ___________ and T_R is __________



Tree Property: complete

Is every **full** tree **complete**?

If every **complete** tree **full**?



Tree ADT

Tree ADT

insert, inserts an element to the tree.

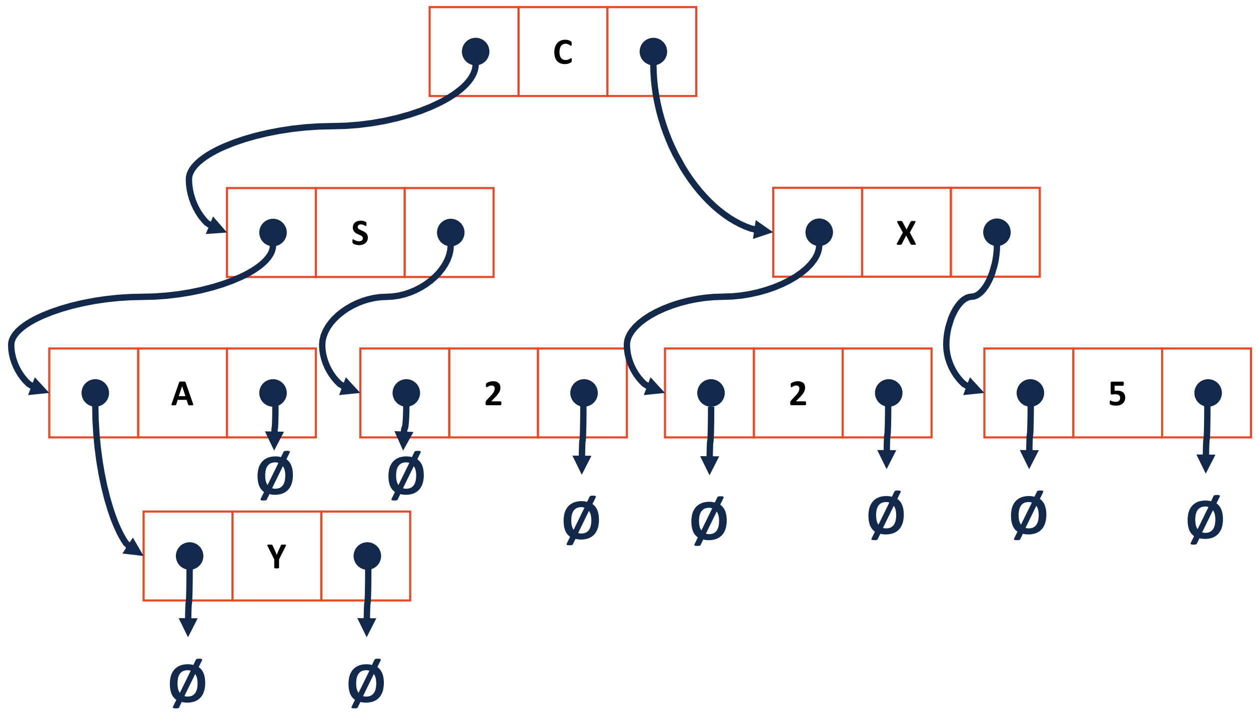
remove, removes an element from the tree.

traverse,

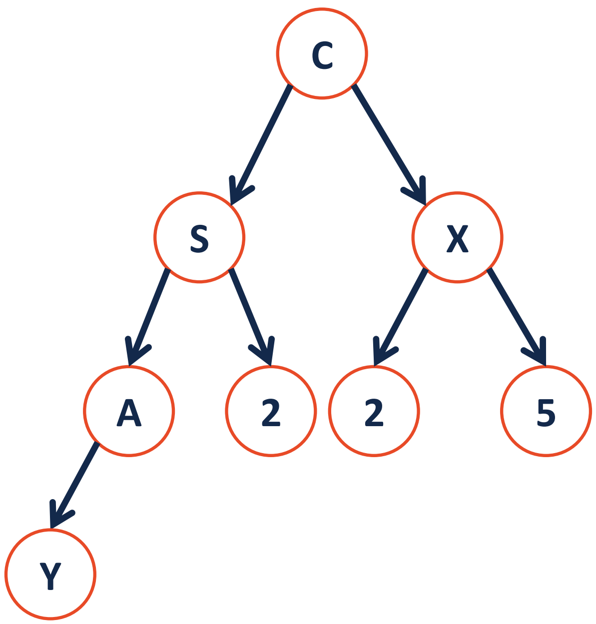
BinaryTree.h

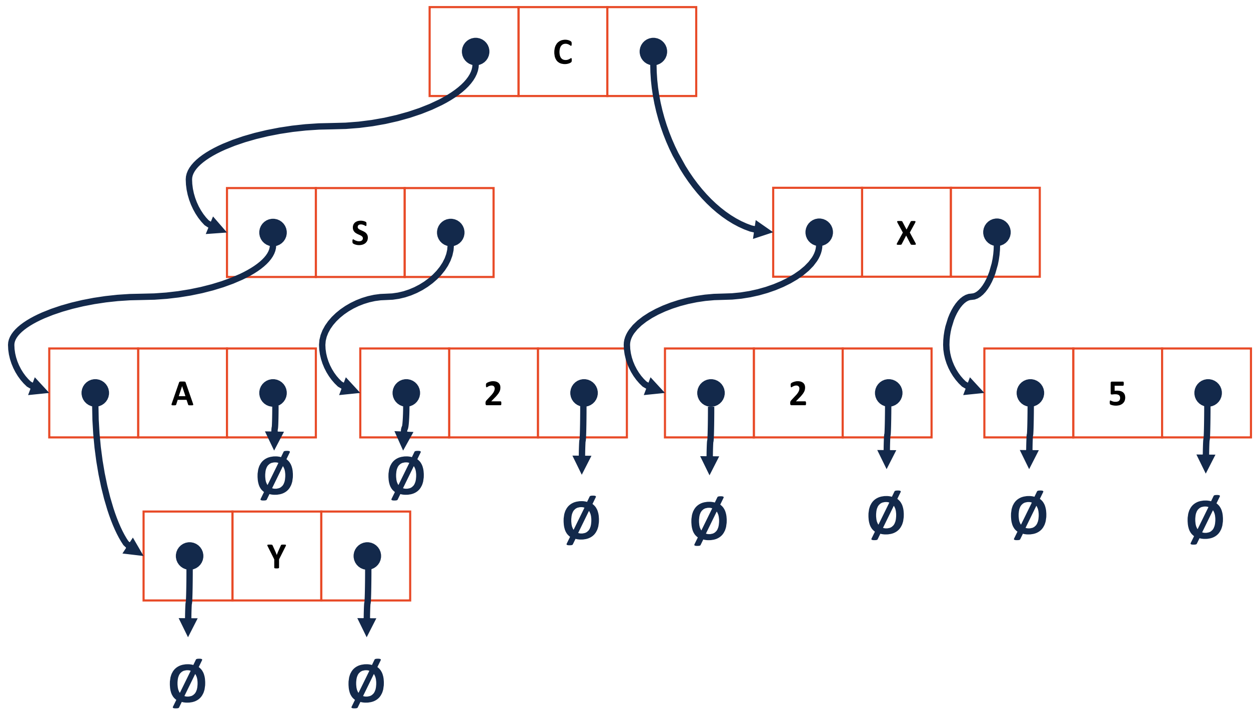
```
#pragma once

template <class T>
class BinaryTree {
  public:
    /* ... */

  private:











};
```

Trees aren’t new:



Trees aren’t new:





How many NULLs?

Theorem: If there are **n** data items in our representation of a binary tree, then there are ____________ NULL pointers.

How many NULLs?

Base Cases:

n = 0:

n = 1:

n = 2:

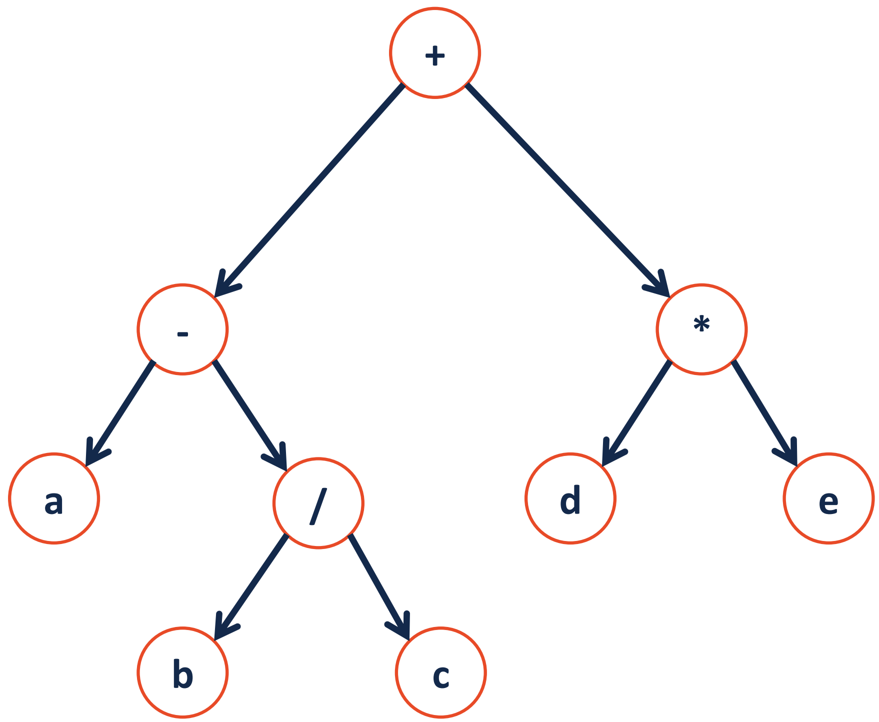
How many NULLs?

Induction Hypothesis:

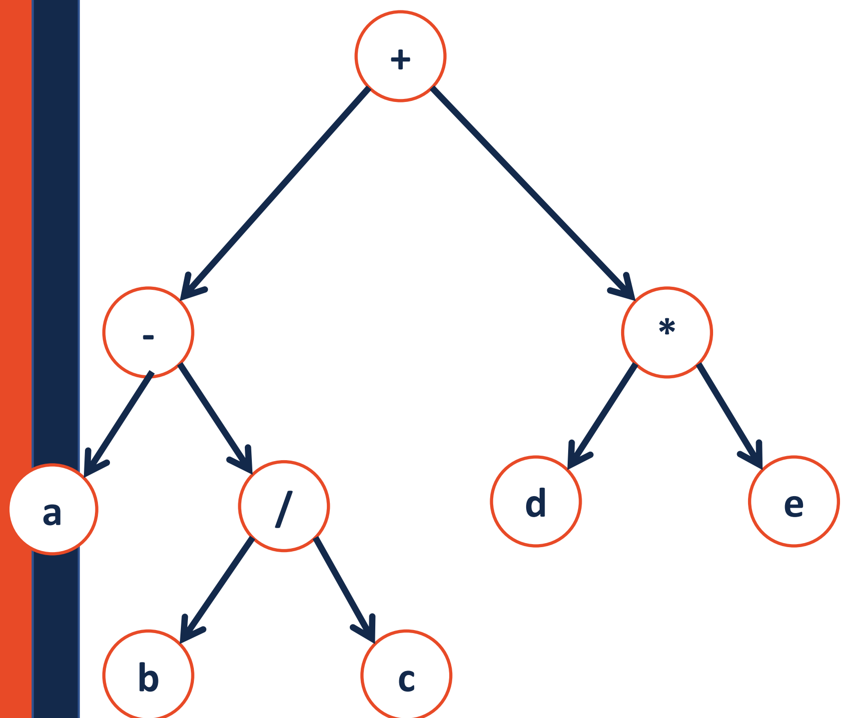
How many NULLs?

Consider an arbitrary tree **T** containing **n** data elements:

Traversals

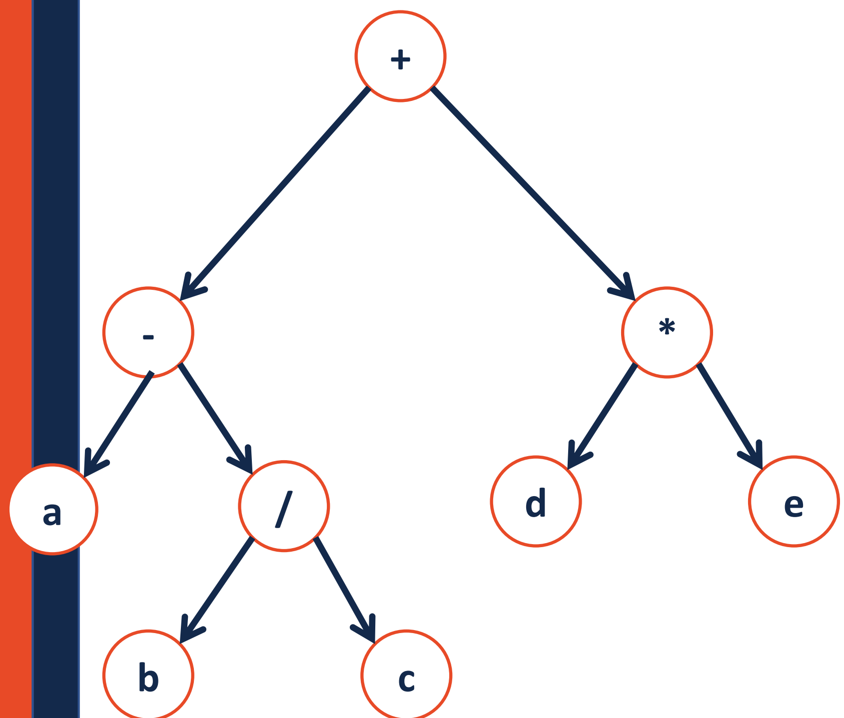


Traversals



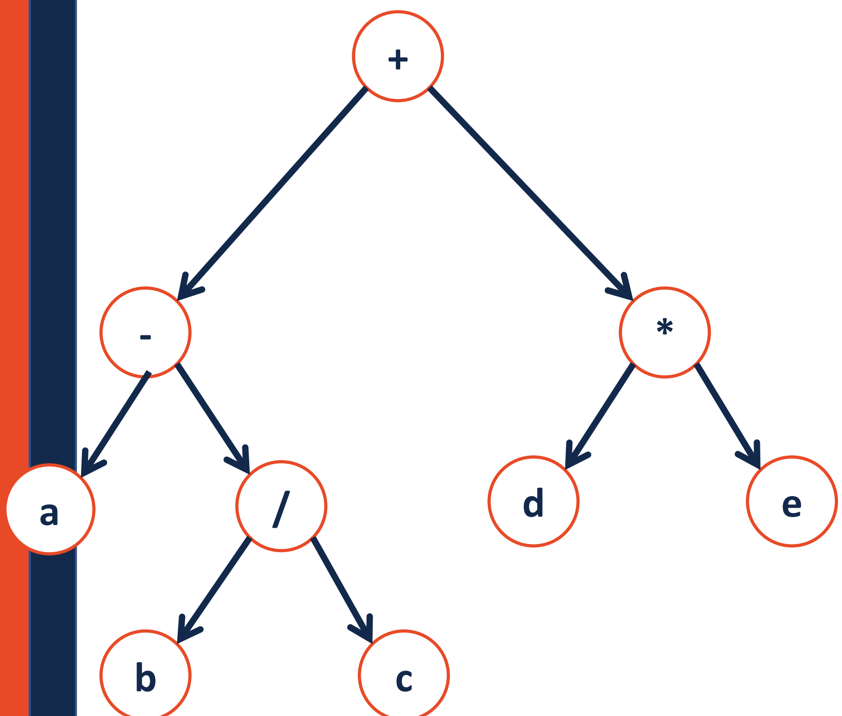

```
template<class T>
void BinaryTree<T>::__Order(TreeNode * root)
{
   if (root != NULL) {
      ______________________;
      ___Order(root->left);
      ______________________;
      ___Order(root->right);
      ______________________;
   }
}
```

Traversals




```
template<class T>
void BinaryTree<T>::__Order(TreeNode * root)
{
   if (root != NULL) {

      ______________________;

      ___Order(root->left);

      ______________________;

      ___Order(root->right);

      ______________________;

   }
}
```

Traversals




```
template<class T>
void BinaryTree<T>::__Order(TreeNode * root)
{
   if (root != NULL) {
      ______________________;
      ___Order(root->left);
      ______________________;
      ___Order(root->right);
      ______________________;
   }
}
```