CS 225

Theory Exam 2 review

TAs: **Mariam Vardishvili**
Patrick Cole
Sayantani Basu

Updates:

0 updates so far.

Review session correction:

1. **BST delete - two child remove:** when removing node n which has both left and right child we are swapping the node with its IOP (in order predecessor), and then calling remove function again for n, so it does not matter weather IOP is a leaf node or not. (You can check full BST remove here: https://github-dev.cs.illinois.edu/cs225-fa18/_lecture/blob/master/16-18-BST/Dictionary.hpp)

Object Lifecycle

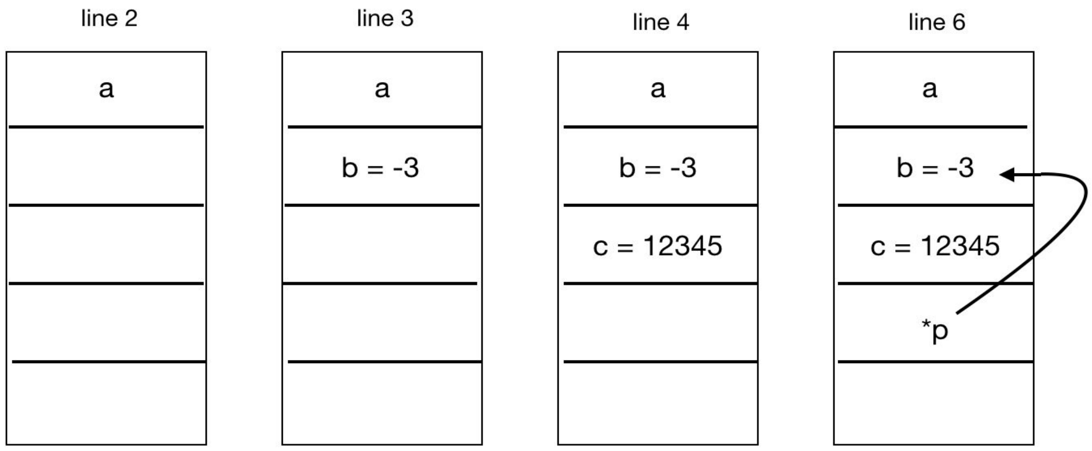
- Lifecycle in stack memory
- Lifecycle in heap memory (new/delete)

Stack Memory

- The default type memory
- Starts at a high address and it grows towards 0
- All variables are by default on stack.

example1.cpp

```
1 int main() {
2   int a;
3   int b = -3;
4   int c = 12345;
5
6   int *p = &b;
7
8   return 0;
9 }
```

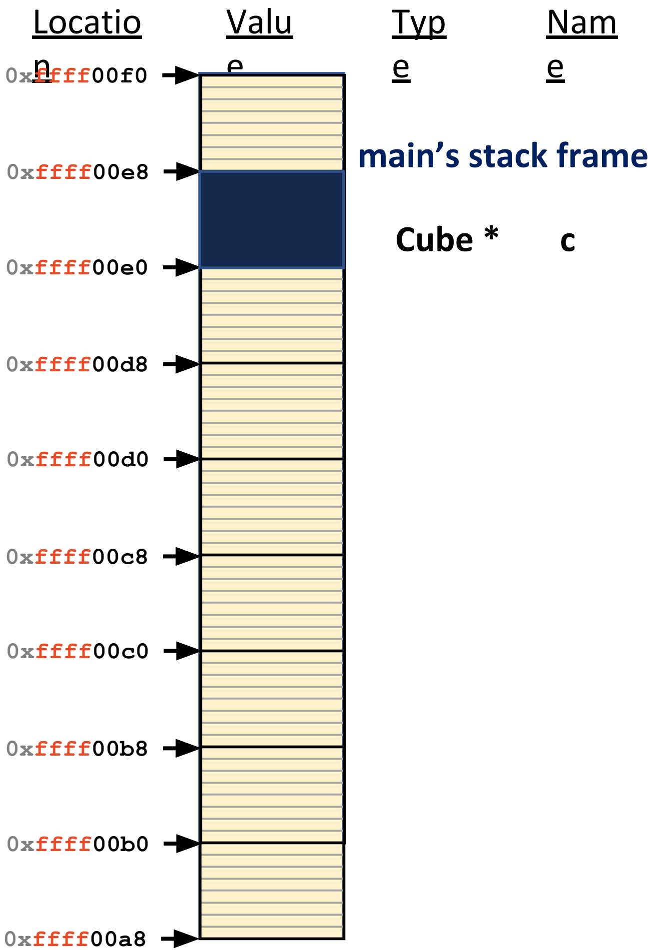


Stack after each line of the code

Stack Frame

- All programs are organized into stack frames.
- A stack frame is created whenever a function is called.
- A stack frame is reclaimed when a function returns (automatically marked as free (not actually freed)). When memory is marked as free, it can be overwritten.

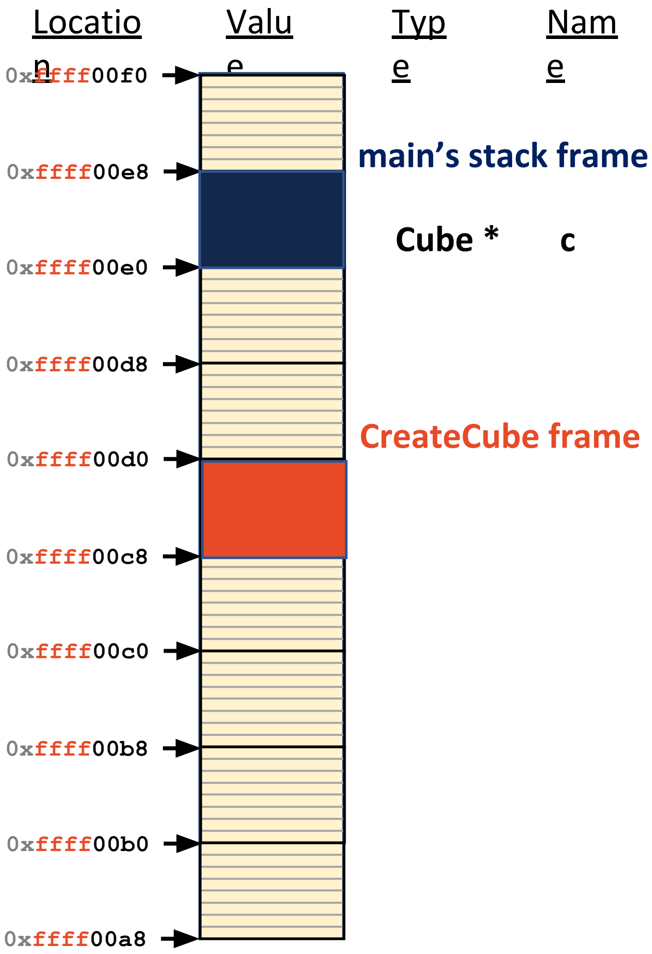
Stack Memory



puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```

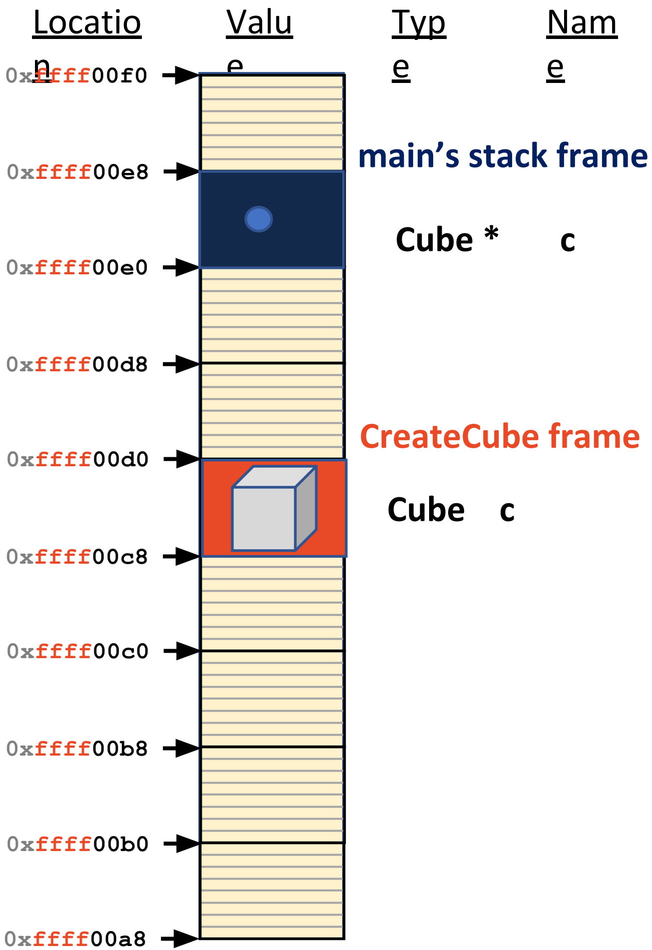
Stack Memory



puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```

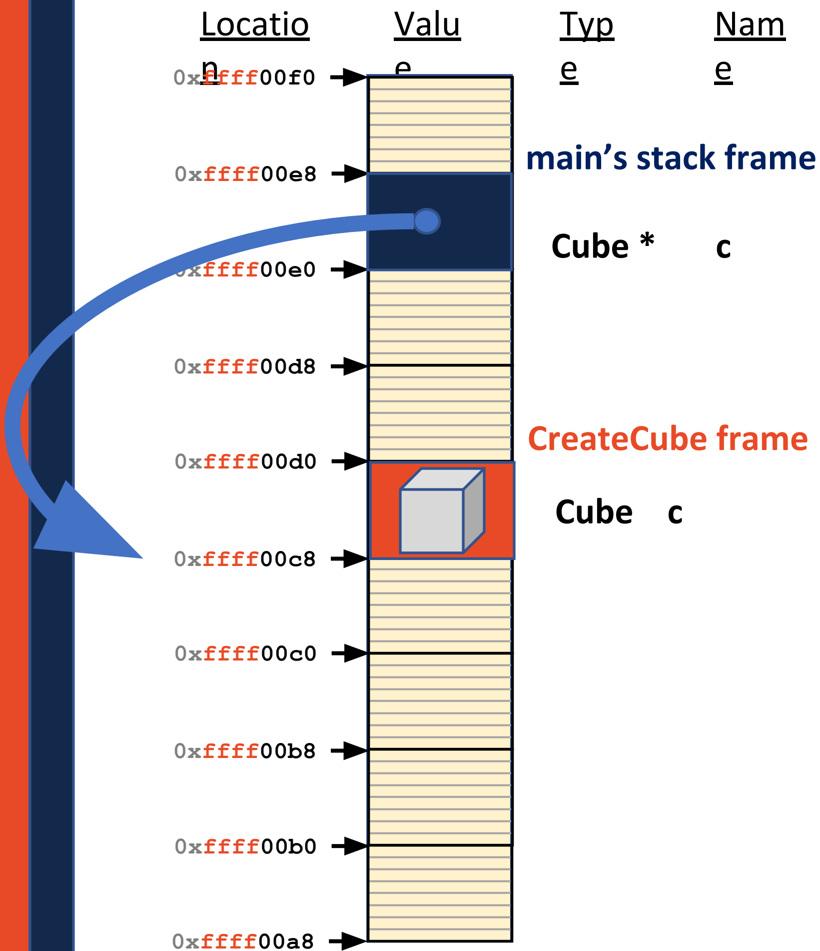
Stack Memory



puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```

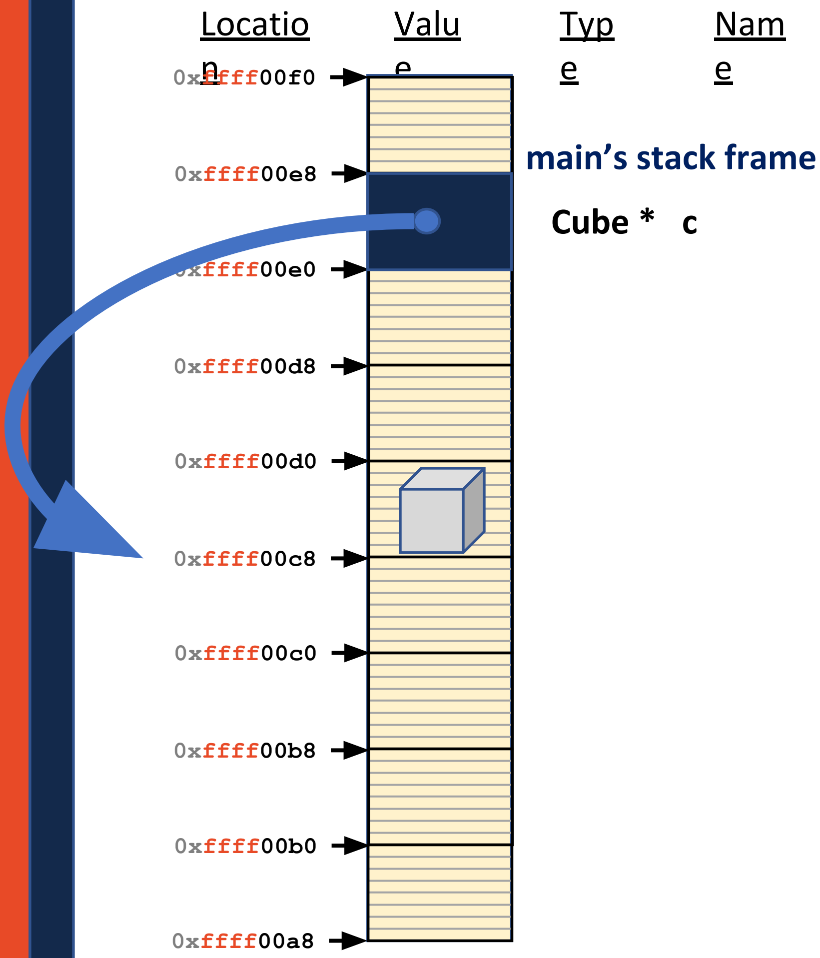
Stack Memory



puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```

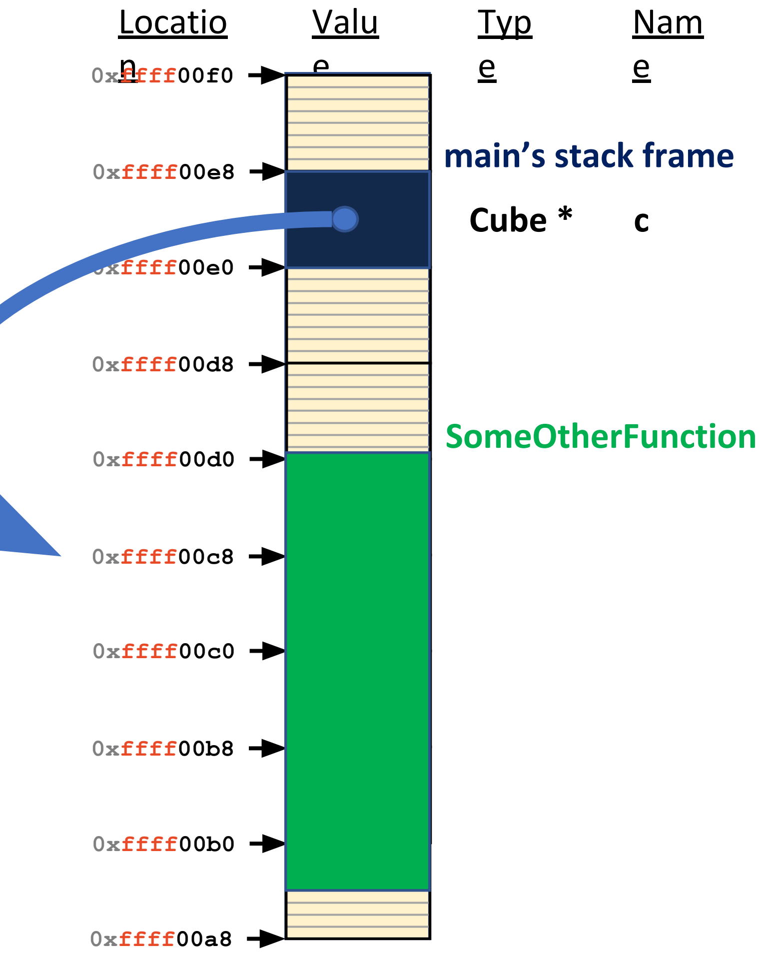
Stack Memory



puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```

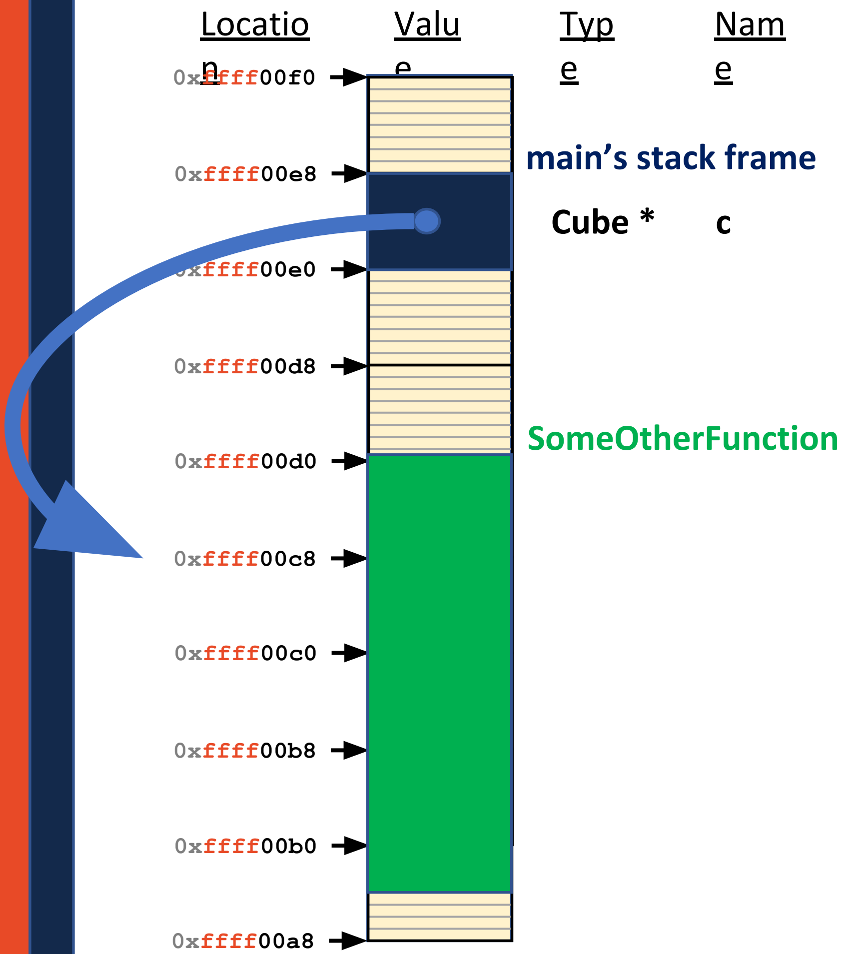
Stack Memory



puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```

Stack Memory



puzzle.cpp

```
#include "Cube.h"
using cs225::Cube;

Cube *CreateCube() {
  Cube c(20);
  return &c;
}

int main() {
  Cube *c = CreateCube();
  SomeOtherFunction();
  double v = c->getVolume();
  double a = c->getSurfaceArea();
  return 0;
}
```

Heap Memory - new

As programmers, we can use heap memory in cases where the lifecycle of the variable exceeds the lifecycle of the function.

The only way to create heap memory is with the use of the **new** keyword. Using **new** will:

1. **Allocate memory**
2. **Construct an object**
3. **Return the pointer**

Heap Memory - delete

2. The <u>only</u> way to free heap memory is with the use of the **delete** keyword. Using **delete** will:

- Destruct the object

- Release memory back to the system

3. Memory is never automatically reclaimed, even if it goes out of scope. Any memory lost, but not freed, is considered to be “leaked memory”.

Worksheet exercises:
lab_debug
lab_memory

Inheritance

- Base class
- Derived class
- Virtual functions
- Pure virtual functions

Inheritance

Classes can be extended to build other classes. We call the class being extended **the base class** and the class inheriting the functionality **the derived class.**

Square.h

```
#pragma once

#include "Shape.h"

class Square : public Shape {
  public:
    double getArea() const;

  private:
    // Nothing!
};
```

Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;

  private:
    double length_;
};
```

Class `Square` extends class `Shape`.

`Square` has access to all public members of `Shape`, but not the private ones.

Square.h

```cpp
#pragma once
#include "Shape.h"

class Square : public Shape {
  public:
    Square();
    Square(double length);
    double getArea() const;

  private:
    // Nothing!
};
```

Square.cpp

```cpp
Square::Square() { }

Square::Square(double length)
: Shape(length) { }

double Square::getArea()
const {
  return
getLength()*getLength();
}
```

Shape.h

```cpp
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;

  private:
    double length_;
};
```

Square.h

```
#pragma once
#include "Shape.h"

class Square : public Shape {
  public:
    Square();
    Square(double length);
    double getArea() const;

  private:
    // Nothing!
};
```

Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;

  private:
    double length_;
};
```

Square.cpp

```
Square::Square() { }

Square::Square(double length)
: Shape(length) { }

double Square::getArea()
const {
  return
    getLength()*getLength();
}
```

…

Square.h

```
#pragma once

#include "Shape.h"

class Square : public Shape {
  public:
```

Square.cpp

```
Square::Square() { }

Square::Square(double length)
: Shape(length) { }

double Square::getArea()
```

`Square::Square(double length): Shape(length) { }`

Syntax for calling parent constructor.

`Square::Square() { }  - version 1`

`Square::Square():Shape() { } - version 2`

Calls `Shape’s` default constructor

When creating child object parent constructor is called first, then child’s constructor.

What if we only have constructor with parameters in the base class?

Derived Classes

[Public Members of the Base Class]:

main.cpp

```
int main() {
  Square sq;
  sq.getLength(); // Returns 1, the length init'd
                  // by Shape's default ctor
  ...
}
```

[Private Members of the Base Class]:

are hidden from derived class!

Polymorphism

Object-Orientated Programming (OOP) concept that a single object may take on the type of any of its base types.

```
Shape *s   = new Shape()

Square *sq = new Square()

Shape*  polyS = new Square()
```

Square.h

```
#pragma once
#include "Shape.h"

class Square : public Shape {
  public:
    Square();
    Square(double length);
    double getArea() const;

  private:
    // Nothing!
};
```

Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;
    void getClass();
  private:
    double length_;
};
```

Square.cpp

```
//...

void Shape::getClass(){
    cout<<“Shape”<<endl;
}

//...
```

```
Shape *s   = new Shape();
   s->getClass();  //prints: Shape
```

```
Square *sq = new Square();
   sq->getClass(); //prints: Shape
```

```
Shape*  polyS = new Square()
   polyS->getClass(); //prints: Shape
```

```
Shape *s   = new Shape();
   s->getClass();  //prints: Shape
```

```
Square *sq = new Square();
   sq->getClass(); //prints: Shape
```

```
Shape*  polyS = new Square()
   polyS->getClass(); //prints: Shape
```

A derived class can have a definition for the member functions of the base class. That base function is said to be overridden.

Square.h

```cpp
#pragma once
#include "Shape.h"

class Square : public Shape {
  public:
    Square();
    Square(double length);
    double getArea() const;
    void getClass();
  private:
    // Nothing!
};
```

Shape.h

```cpp
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;
    void getClass();
  private:
    double length_;
};
```

Square.cpp

```cpp
//...

void Square::getClass(){
    cout<<“Square”<<endl;
}


void Shape::getClass(){
    cout<<“Shape”<<endl;
}

//...
```

```
Shape *s  = new Shape();
  s->getClass();  //prints: Shape
```

```
Square *sq = new Square();
  sq->getClass(); //prints: Square
```

```
Shape*  polyS = new Square()
  polyS->getClass();
```

```
Shape *s   = new Shape();
   s->getClass();  //prints: Shape
```

```
Square *sq = new Square();
   sq->getClass(); //prints: Square
```

```
Shape*  polyS = new Square()
   polyS->getClass(); //prints: Shape
```

```
Shape*  polyS = new Square()

   polyS->getClass(); //prints: Shape
```

How to fix this:

A **virtual** function a member function which is declared within base class and is re-defined (Overriden) by derived class.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.

Functions are declared with a **`virtual`** keyword in base class

```
Shape*  polyS = new Square()
   polyS->getClass(); //prints: Square
```

```
Shape*  polyS = new Square()

   polyS->getClass(); //prints: Square
```

A virtual function

1. Must be declared in public section of class.
2. The prototype of virtual functions should be same in base as well as derived class.
3. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.

Square.h

```
#pragma once
#include "Shape.h"

class Square : public Shape {
  public:
    Square();
    Square(double length);
    double getArea() const;
    void getClass();
  private:
    // Nothing!
};
```

Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;
    virtual void getClass();
  private:
    double length_;
};
```

Square.cpp

```
//...

void Square::getClass(){
    cout<<“Square”<<endl;
}


void Shape::getClass(){
    cout<<“Shape”<<endl;
}

//...
```

Pure virtual function

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation.

Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;
    virtual int nOfEdges();
  private:
    double length_;
};
```

Pure virtual function

Pure virtual functions is a virtual function which is only declared, without any implementation. `virutal void function()=0;`

Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;
    virtual int nOfEdges()=0;
  private:
    double length_;
};
```

Abstract Class:

[Requirement]:

Has at least one pure virtual function;

(No other constraints)

[Syntax]:

No special keywords.

[As a result]:

Cannot create an object of the abstract class;

Derived classes should override pure virtual functions, or they will also become abstract classes.

Square.h

```
#pragma once
#include "Shape.h"

class Square : public Shape {
  public:
    Square();
    Square(double length);
    double getArea() const;

  private:
    // Nothing!
};
```

Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;
    virtual int nOfEdges()=0;
  private:
    double length_;
};
```

`Shape` is an abstract class, because it has pure virtual function.

`Square` is an abstract class, since it has inherited pure virtual function and it has no implementation for it.

Square.h

```
#pragma once
#include "Shape.h"

class Square : public Shape {
  public:
    Square();
    Square(double length);
    double getArea() const;
    int nOfEdges();
  private:
    // Nothing!
};
```

Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;
    virtual int nOfEdges()=0;
  private:
    double length_;
};
```

Square.cpp

```
//...
 int Square::nOfEdges(){
   return 4;
 }

//...
```

…

```
Shape *s   = new Shape();
```

ERROR! Shape is an abstract class!

~~cout<< s->nOfEdges();~~

```
Square *sq = new Square();
cout<< sq->nOfEdges(); //prints: 4
```

```
Shape*  polyS = new Square()
polyS-> nOfEdges(); //prints: 4
```

Square.h

```
#pragma once
#include "Shape.h"

class Square : public Shape {
  public:
    Square();
    Square(double length);
    double getArea() const;
    int nOfEdges();
  private:
    Double area_
};
```

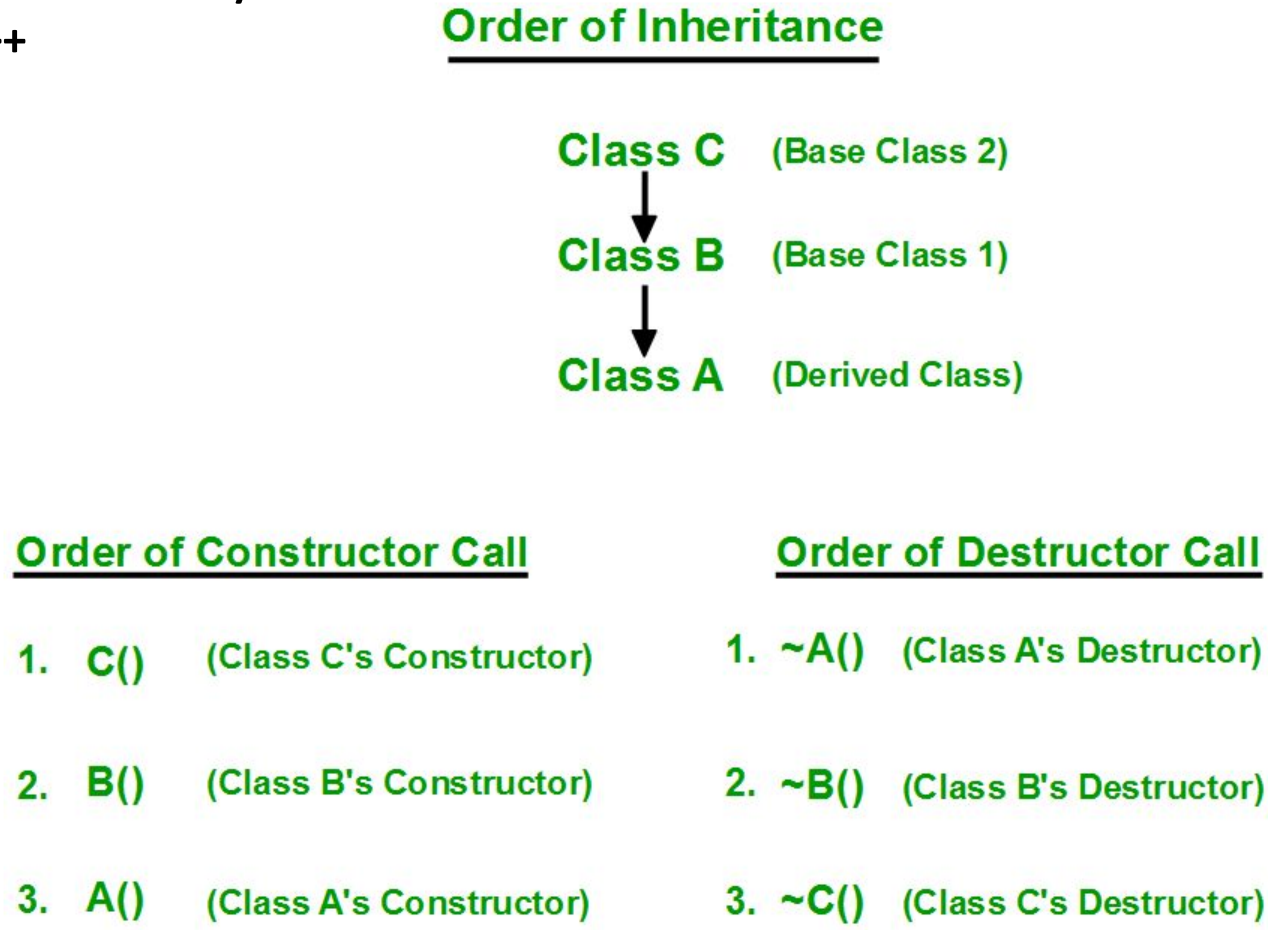
Shape.h

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;
    virtual int nOfEdges()=0;
  private:
    double length_;
};
```

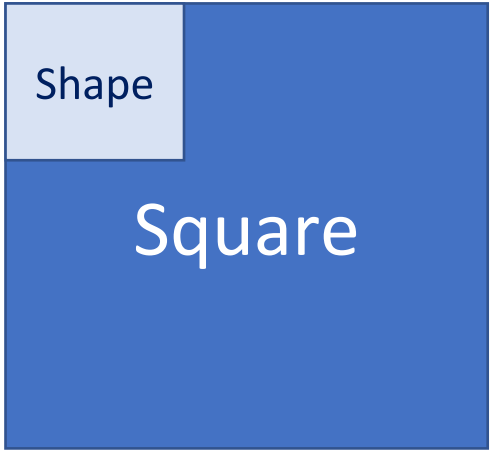
```
Shape *s  = new Square();
cout<<s->area_;
```

ERROR!!

Order of Constructor/ Destructor Call in C++



https://www.geeksforgeeks.org/order-constructor-destructor-call-c/

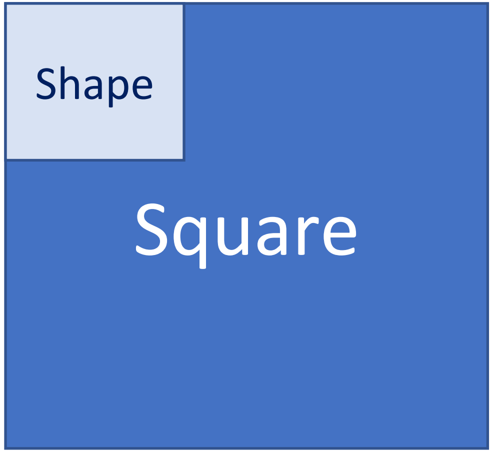


Creating object of `Square` means that you are creating object of `Shape` and `Square`.

`Square *s = new Square();` calls `Square's` constructor, which calls `Shape's` constructor before its body is executed;

```
Square::Square(double length): Shape(length) { }

Square::Square():Shape() { }
```

When creating child object parent constructor is called first, then child's constructor is executed.



Destructors are called in the opposite order of that of Constructors

`delete s;` calls `Square's` destructor, which calls `Shape's` destructor after its body is executed;

```
Shape *s = new  Square();
delete s;
```

In order to destruct the `Square` object properly, when we are using parent pointer, destructor in `Shape` class must be `virtual`, otherwise only destructor of parent class will be called;

```cpp
class Shape {
public:
 Shape(){
    cout<<"Ctor Shape"<<endl;
  }
  ~Shape(){
    cout<<"Dtor Shape"<<endl;
  }
};

```

```cpp
class Square : public Shape{
public:
  int *i;
  Square(): Shape(){
    cout<<"Ctor Square "<<endl;
    i = new int;
  }
  ~Square(){
    cout<<“Dtor Square"<<endl;
    if(i != NULL)
      delete i;
    i = NULL;
  }
};
```

```
int main(){
  Shape *b = new Square();
  Square *c = new Square();

  delete b;
  delete c;
  return 0;
}
```

```
int main(){
  Shape *b = new Square();
  Square *c = new Square();

  delete b;
  delete c;
  return 0;
}
```

Ctor Shape

Ctor Square

```
int main(){
   Shape *b = new Square();
   Square *c = new Square();

   delete b;
   delete c;
   return 0;
}
```

Ctor Shape
Ctor Square
Ctor Shape
Ctor Square

```
int main(){
   Shape *b = new Square();
   Square *c = new Square();

   delete b;
   delete c;
   return 0;
}
```

```
class Shape {
public:
 Shape(){
     cout<<“Ctor Shape"<<endl;
   }
  ~Shape(){
     cout<<"Dtor Shape"<<endl;
  }
};
```

Ctor Shape
Ctor Square
Ctor Shape
Ctor Square
Dtor Shape

Since `~Shape` is not virtual and `b` is base class pointer, only `Shape` destructor will be called;

```cpp
int main(){
   Shape *b = new Square();
   Square *c = new Square();

   delete b;
   delete c;
   return 0;
}
```

Ctor Shape
Ctor Square
Ctor Shape
Ctor Square
Dtor Shape
Dtor Square
Dtor Shape

```cpp
class Shape {
public:
 Shape(){
     cout<<"Ctor Shape"<<endl;
   }
   ~Shape(){
     cout<<"Dtor Shape"<<endl;
   }
};
```

```cpp
class Square : public Shape{
public:
  int *i;
  Square(): Shape(){
    cout<<"Ctor Square "<<endl;
    i = new int;
  }
  ~Square(){
    cout<<“Dtor Square"<<endl;
    if(i != NULL)
      delete i;
    i = NULL;
  }
};
```

```
class Shape {
public:
 Shape(){
    cout<<"Ctor Shape"<<endl;
  }
 virtual ~Shape(){
    cout<<"Dtor Shape"<<endl;
  }
};

```

```
class Square : public Shape{
public:
  int *i;
  Square(): Shape(){
    cout<<"Ctor Square "<<endl;
    i = new int;
  }
  ~Square(){
    cout<<“Dtor Square"<<endl;
    if(i != NULL)
      delete i;
    i = NULL;
  }
};
```

```
int main(){
  Shape *b = new Square();
  Square *c = new Square();

  delete b;
  delete c;
  return 0;
}
```

```
int main(){
  Shape *b = new Square();
  Square *c = new Square();

  delete b;
  delete c;
  return 0;
}
```

Ctor Shape

Ctor Square

```
int main(){
   Shape *b = new Square();
   Square *c = new Square();

   delete b;
   delete c;
   return 0;
}
```

Ctor Shape
Ctor Square
Ctor Shape
Ctor Square

```
int main(){
  Shape *b = new Square();
  Square *c = new Square();

  delete b;
  delete c;
  return 0;
}
```

```
class Shape {
public:
 Shape(){
    cout<<“Ctor Shape"<<endl;
  }
virtual ~Shape(){
    cout<<"Dtor Shape"<<endl;
  }
};
```

Ctor Shape
Ctor Square
Ctor Shape
Ctor Square
Dtor Square
Dtor Shape

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.

```cpp
int main(){
   Shape *b = new Square();
   Square *c = new Square();

   delete b;
   delete c;
   return 0;
}
```

Ctor Shape
Ctor Square
Ctor Shape
Ctor Square
Dtor Square
Dtor Shape
Dtor Square
Dtor Shape

Useful material:

Worksheet
lab_inheritance

Templates in C++

```

Template <typename T>
T maximum(T a, T b) {
  T result;
  result = (a > b) ? a : b;
  return result;
}
```

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

```
maximum(1,2); //ints
maximum(3.4, 2.8); //doubles
maximum(Cube c1(7), Cube c2(8)); //user-defined classes
```

List ADT

List ADT	Definition of Functionality
Create the empty list	Creates and empty list.
Add data to the list	Store data.
Get data from the list	Access data.
Remove data from the list	Remove data.
Check if a list is empty/size	How much data is in the list.

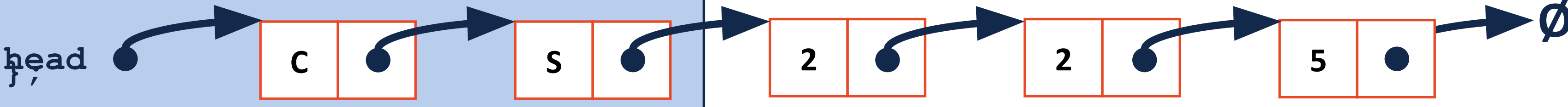
There are two basic implementations of the list:
Array - sequential block of items.
Linked Memory - linked list.

Linked List

- Operation **insertAtFront**, including running time and insertion strategies
- Operation **insertAtIndex**, including running time, on both a sorted and unsorted list
- Operation **removeAtIndex**, including running time, on both a sorted and unsorted list
- Operation **insertAfterElement**, including running time, on both a sorted and unsorted list
- Operation **removeAfterElement**, including running time, on both a sorted and unsorted list
- Operation **findIndex**, including running time, on both a sorted and unsorted list
- Operation **findData**, including running time, on both a sorted and unsorted list

List.h

```
#pragma once

template <typename T>
class List {
  public:
    /* ... */

  private:
    class ListNode {
      public:
        T & data;
        ListNode * next;
        ListNode(T & data) :
        data(data), next(NULL) {
}
    };

    ListNode *head_;

};
```



A node has two member variables:

A list node pointer that points to the next block (`ListNode * next`).

The data stored in the block (`T& data`)

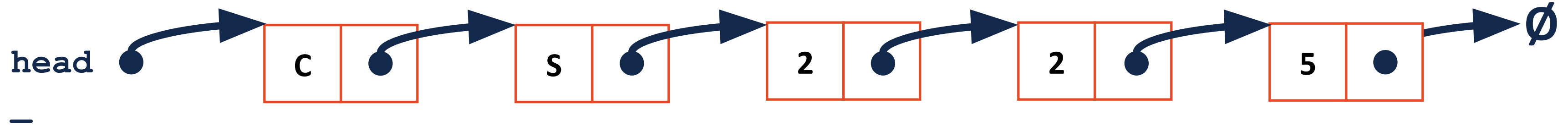
List.hpp

```cpp
#include "List.h"

template <typename T>
void List::insertAtFront(const T& t) {
   ListNode * node = new ListNode(t);
   node->next = head_;
   head_ = node;

}
```

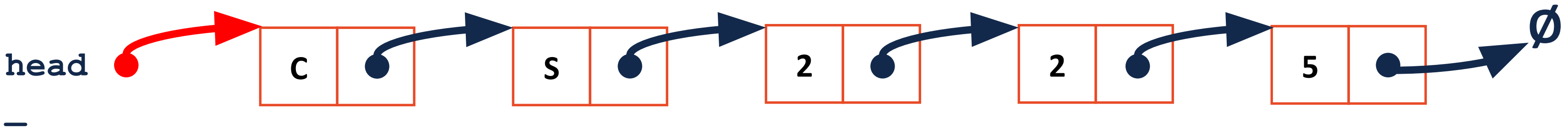
Running time of the `insertAtFront` is $O(1)$

List.hpp

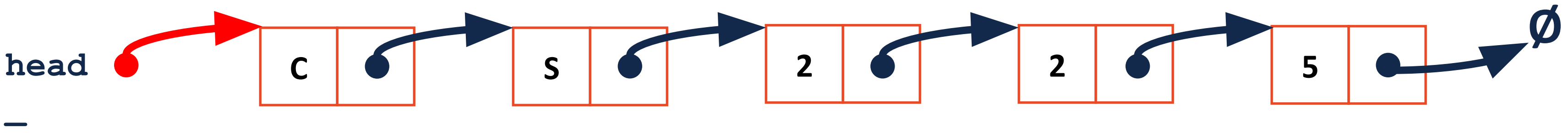
```
// Recursive Solution:
template <typename T>
typename List<T>::ListNode *& List<T>::_index(unsigned index) {
   //return a reference to a ListNode pointer
   return _index_helper(index, _head);
}

ListNode *& _index_helper(unsigned index, ListNode *& node) {
  if (index == 0) {
    return node;
  } else {
    return _index(index - 1, node -> next);
}
// shift one in each recursive call
```



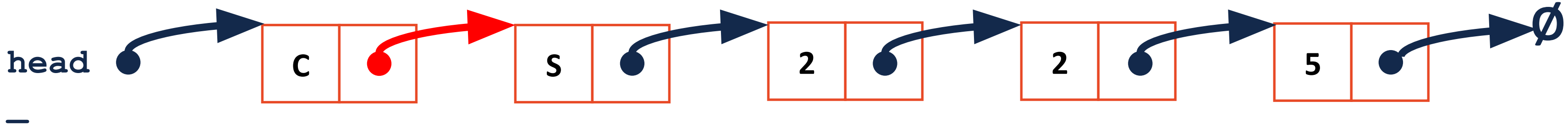
1. Index = 2 and **ListNode *& node**



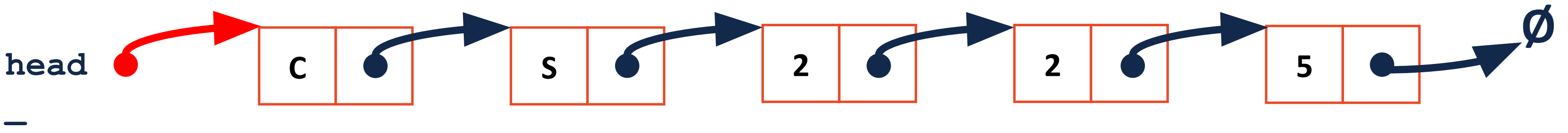
1. `Index = 2` and **`ListNode *& node`**



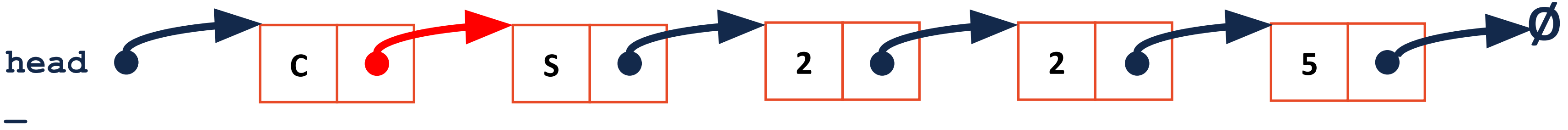
2. `Index = 1` and **`ListNode *& node`**



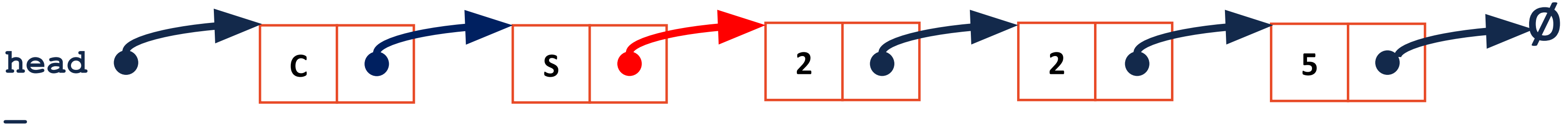
1. `Index = 2` and **`ListNode *& node`**



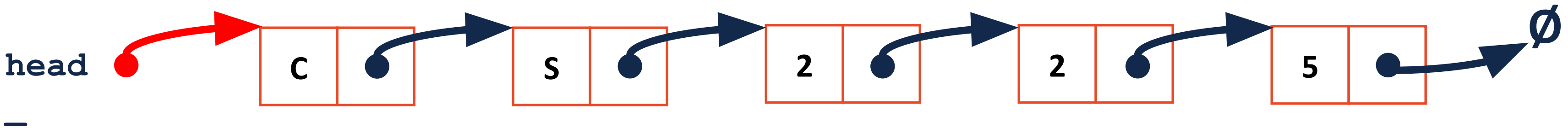
2. `Index = 1` and **`ListNode *& node`**



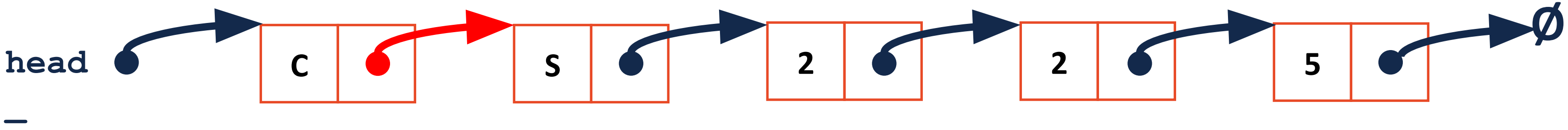
3. `Index = 0` and **`ListNode *& node`**



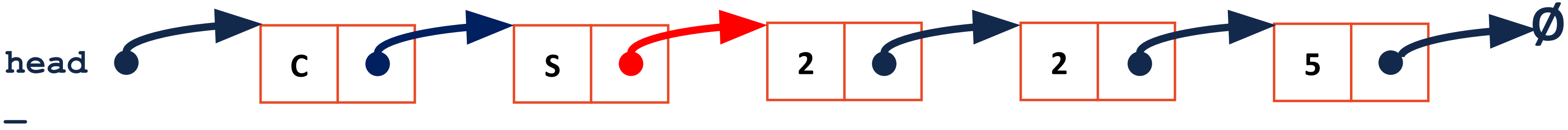
1. `Index = 2` and **`ListNode *& node`**



2. `Index = 1` and **`ListNode *& node`**



3. `Index = 0` and **`ListNode *& node`**

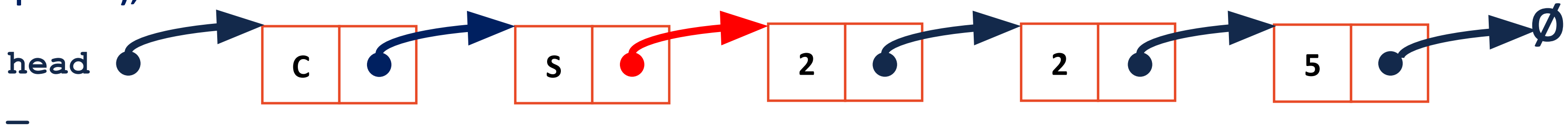


Running time O(n)

List.cpp

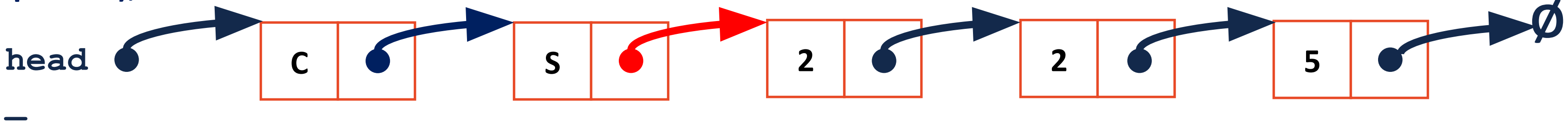
```
template <typename T>
void List<T>::insert(const T & t, unsigned index)
{
  ListNode *& node = _index(index);
  ListNode * newNode = new ListNode(t);
  newNode -> next = node;
  node = newNode;
}
```

`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);



```
template <typename T>
void List<T>::insert(const T & t, unsigned index)
{
  ListNode *& node = _index(index);
  ListNode * newNode = new ListNode(t);
  newNode -> next = node;
  node = newNode;
}
```

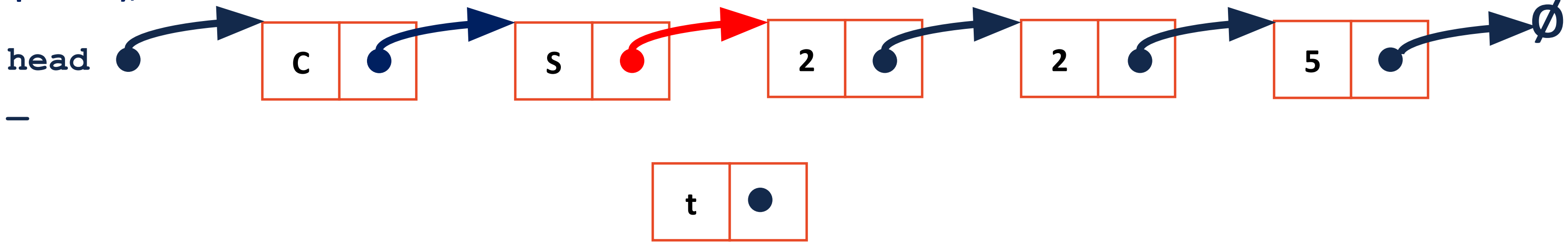
`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);



List.cpp

```cpp
template <typename T>
void List<T>::insert(const T & t, unsigned index)
{
  ListNode *& node = _index(index);
  ListNode * newNode = new ListNode(t);
  newNode -> next = node;
  node = newNode;
}
```

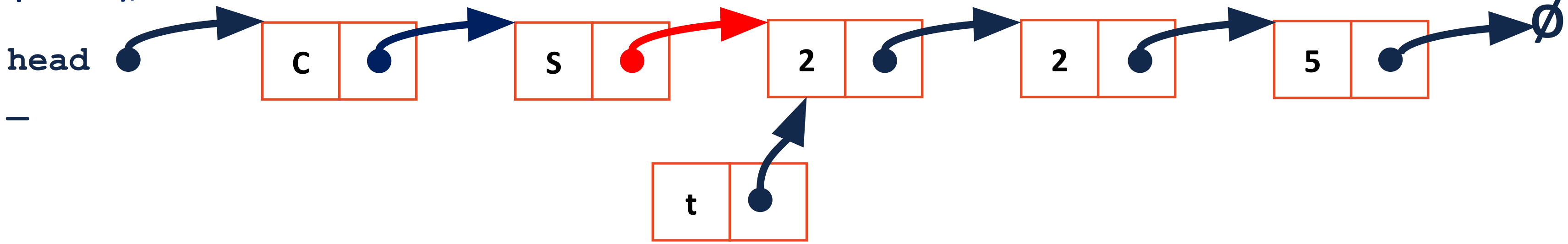
`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);



List.cpp

```cpp
template <typename T>
void List<T>::insert(const T & t, unsigned index)
{
  ListNode *& node = _index(index);
  ListNode * newNode = new ListNode(t);
  newNode -> next = node;
  node = newNode;
}
```

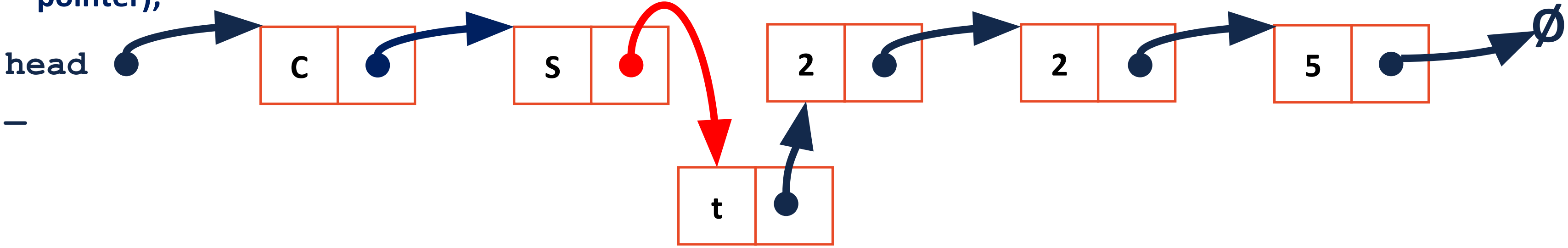
`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);



List.cpp

```
template <typename T>
void List<T>::insert(const T & t, unsigned index)
{
  ListNode *& node = _index(index);
  ListNode * newNode = new ListNode(t);
  newNode -> next = node;
  node = newNode;
}
```

`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);



List.cpp

```
template <typename T>
void List<T>::insert(const T & t, unsigned index)
{
  ListNode *& node = _index(index);
  ListNode * newNode = new ListNode(t);
  newNode -> next = node;
  node = newNode;
}
```

Running time of `_index(index)` is `O(n)`.

List.cpp

```
template <typename T>
void List<T>::insert(const T & t, unsigned index)
{
  ListNode *& node = _index(index);
  ListNode * newNode = new ListNode(t);
  newNode -> next = node;
  node = newNode;
}
```

Running time of `_index(index)` is `O(n)`.

Adding new element after given pointer: `O(1)`

List.cpp

```
template <typename T>
void List<T>::insert(const T & t, unsigned index)
{
   ListNode *& node = _index(index);
   ListNode * newNode = new ListNode(t);
   newNode -> next = node;
   node = newNode;
}
```

Running time of `_index(index)` is O(n).

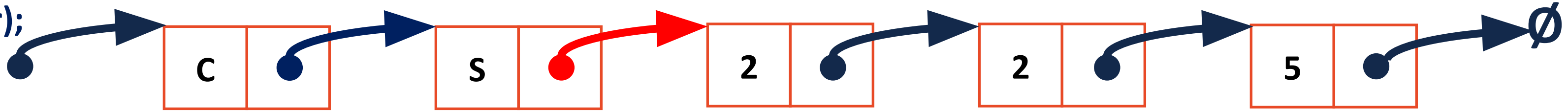
Adding new element after given pointer: O(1)

Running time of `insert(...)` is O(n)

```cpp
template <typename T>
void & List<T>::remove(unsigned index) {
  ListNode *& node = _index(index);
  ListNode * temp  = node;
  node = node -> next;
  delete temp;
}
```

`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);

head

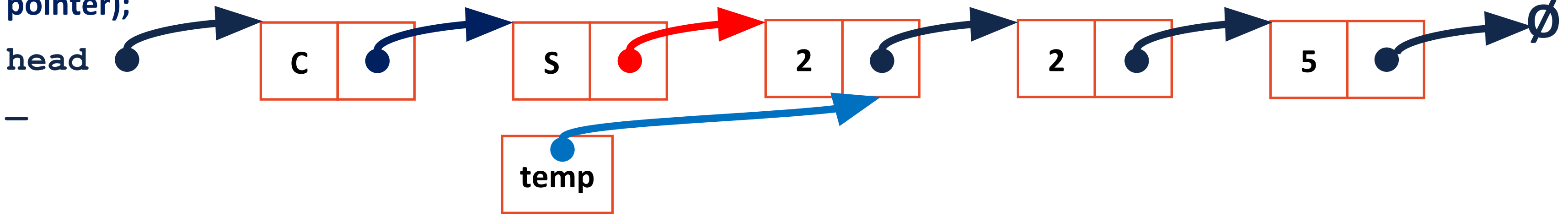


_

List.cpp

```
template <typename T>
void & List<T>::remove(unsigned index) {
  ListNode *& node = _index(index);
  ListNode * temp  = node;
  node = node -> next;
  delete temp;
}
```

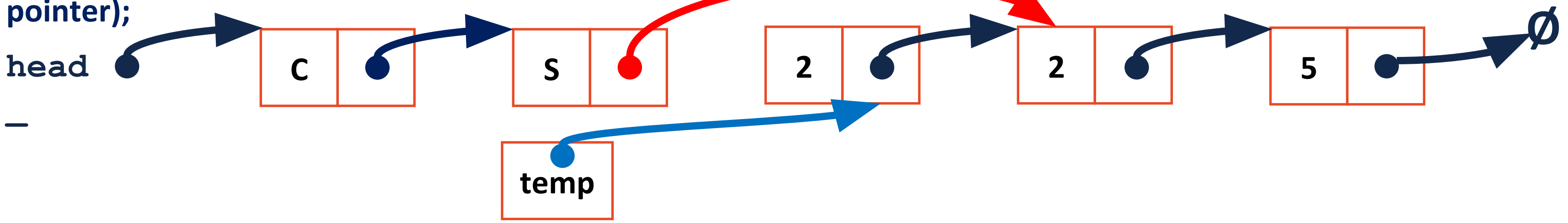
`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);



List.cpp

```
template <typename T>
void & List<T>::remove(unsigned index) {
  ListNode *& node = _index(index);
  ListNode * temp  = node;
  node = node -> next;
  delete temp;
}
```

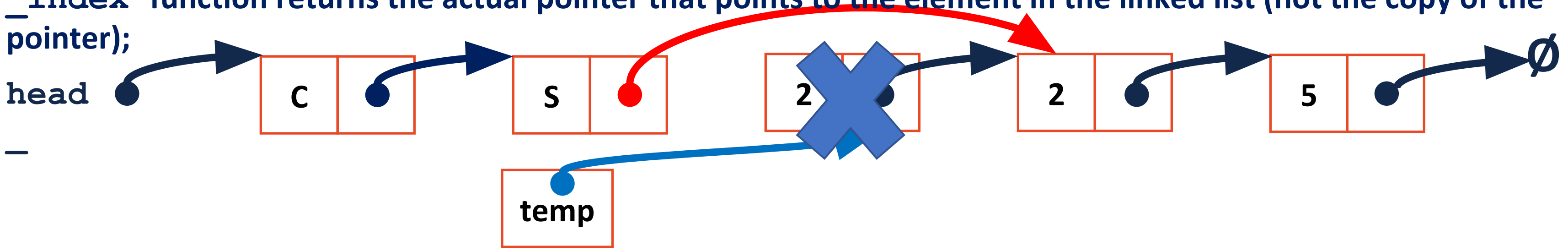
`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);



List.cpp

```
template <typename T>
void & List<T>::remove(unsigned index) {
  ListNode *& node = _index(index);
  ListNode * temp  = node;
  node = node -> next;
  delete temp;
}
```

_index function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);



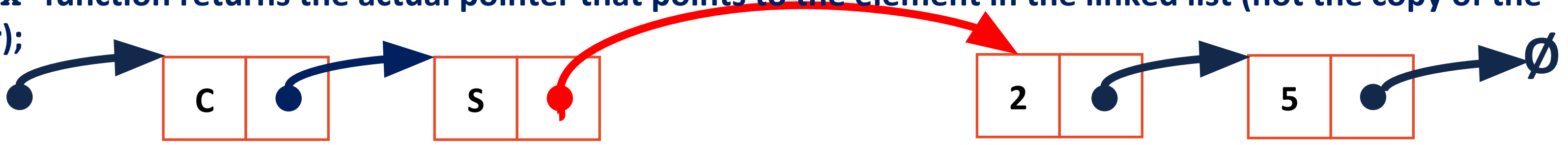
List.cpp

```cpp
template <typename T>
void & List<T>::remove(unsigned index) {
  ListNode *& node = _index(index);
  ListNode * temp  = node;
  node = node -> next;
  delete temp;
}
```

`_index` function returns the actual pointer that points to the element in the linked list (not the copy of the pointer);

head

_



List.cpp

```
template <typename T>
void & List<T>::remove(unsigned index) {
  ListNode *& node = _index(index);
  ListNode * temp  = node;
  node = node -> next;
  delete temp;
}
```

Running time of `_index(index)` is `O(n)`.

Reassigning given pointer and deleting node : `O(1)`

Running time of `remove(...)` is `O(n)`

Array List

- Operation **insertAtFront**, including running time, resize strategies, and proofs
- Operation **insertAtIndex**, including running time, on both a sorted and unsorted list
- Operation **removeAtIndex**, including running time, on both a sorted and unsorted list
- Operation **insertAfterElement**, including running time, on both a sorted and unsorted list
- Operation **removeAfterElement**, including running time, on both a sorted and unsorted list
- Operation **findIndex**, including running time, on both a sorted and unsorted list
- Operation **findData**, including running time, on both a sorted and unsorted list

Array list Implementation

Store everything in an array

C	S	2	2	5
[0]	[1]	[2]	[3]	[4]

```
#pragma once

template <typename T>
class List {
  public:
      /* ... */
  private:
    T* arr_; // the content array
    unsigned count_; // the maximum size possible; the allocated array
size
    unsigned capacity; //the size in use; the number of current elements

};
```

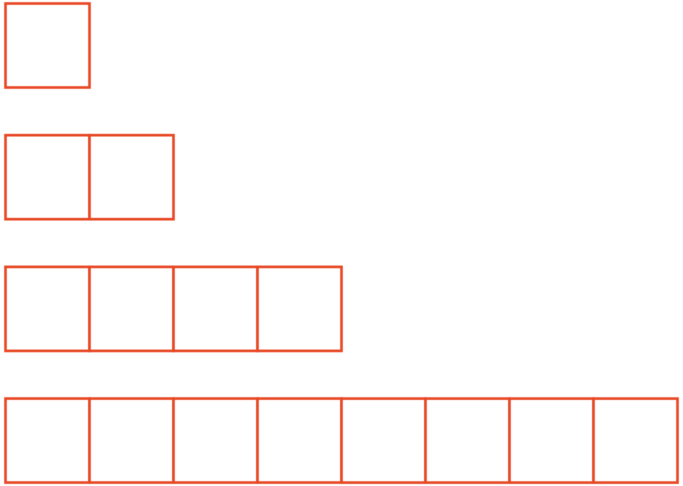
Array list Implementation

Store everything in an array

C	**S**	**2**	**2**	**5**
[0]	[1]	[2]	[3]	[4]

To insert new element in the array we need to resize it.

Resize Strategy – Details

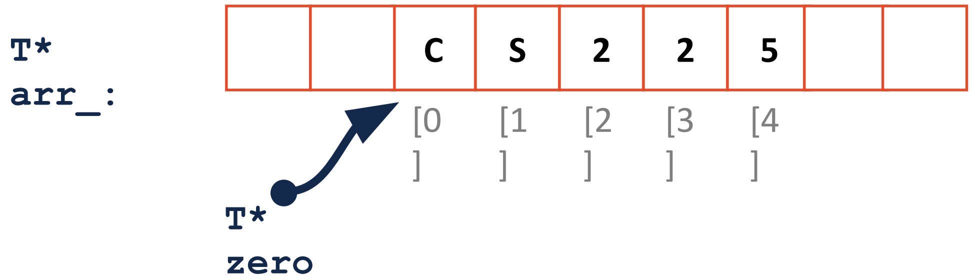


Total number of copies to grow an array from 0 to n is: **2n-1 = O(n)**

Total number of inserts as we grow the array: n

Running time per insert: $\frac{O(n)}{n} = O(1)^*$

Array list Implementation




```
#pragma once

template <typename T>
class List {
  public:
     /* ... */
  private:
    T* arr_; // the content array
    T* zero; //pointer to the first element in the array
    unsigned count_; // the maximum size possible; the allocated array size
    unsigned capacity; //the size in use; the number of current elements

};
```

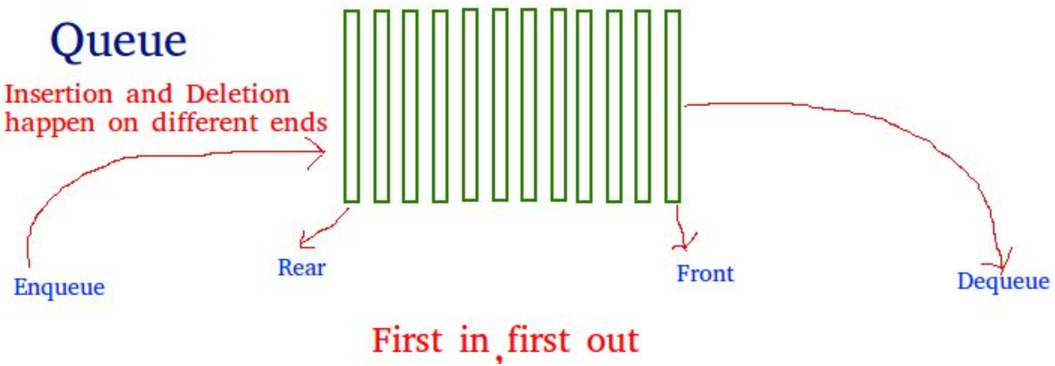
	Singly Linked List	Array list
Insert/Remove at **front**	O(1)	O(1)*
Insert after a **given** element	O(1)	O(n)
Remove after a **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n) Must reach the index	O(n)
Remove at **arbitrary** location	O(n) Must reach the index	O(n)

Copy until we reach an end to make space

Stacks and Queues

- Using a list to build a stack/queue
- Operations pop, push, enqueue, and dequeue, including running times
- Applications of stacks and queues (*see lab_quack*)

Queue ADT



- [Order]:

First in First out;

- [Operations]:

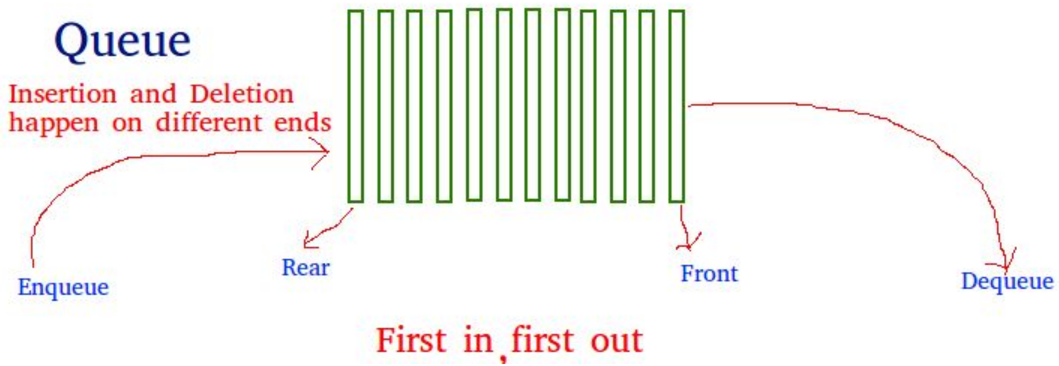
Enqueue – add element in the end

Dequeue – delete element from front

- [Implementation]:

Removing element from the beginning and adding element in the back takes O(1) time using List (linked list, array impl.).

Queue ADT



- [Order]:

First in First out;

- [Operations]:

Enqueue – add element in the end

Dequeue – delete element from front

- [Implementation]:

Removing element from the beginning and adding element in the back takes O(1) time using List (linked list, array impl.).
For linked list we should also save pointer to the last element of the list, so that we can insert in the end in O(1) time.

Queue.h

```cpp
#pragma once

template <typename T>
class Queue {
  public:
    void enqueue(T e);
    T dequeue();
    bool isEmpty();

  private:
    T *items_; //array pointer
    unsigned capacity_;
    unsigned count_;
    unsigned start_; // start
// index of the queue, used to
// avoid shifting data.
};
```

What type of implementation is this Queue?

Array List

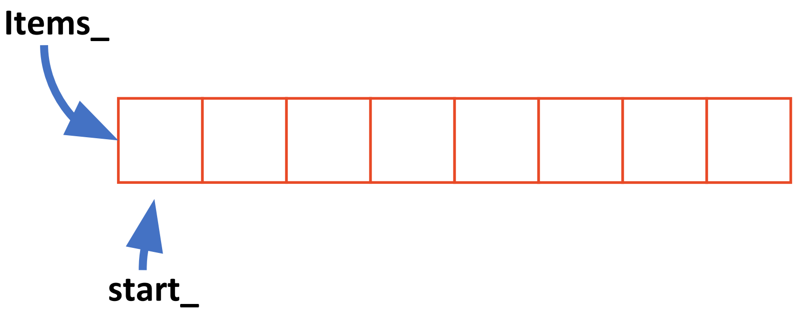
How is the data stored on this Queue?

By value, since array stores type T

Keep `start_` as a pointer to the beginning of the queue, move `start_` to the left when removing element from front.

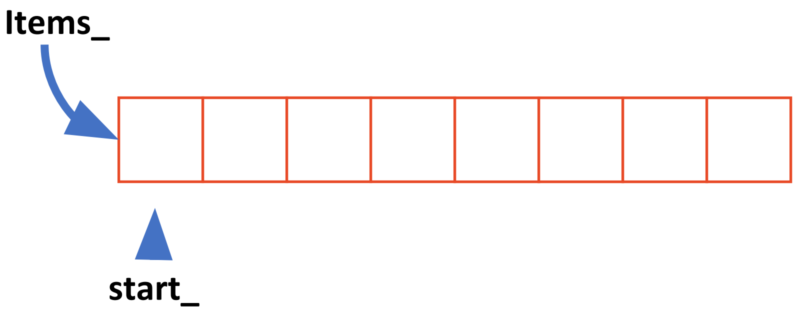
Calculate insertion index: `(start_ + count_)% capacity`

When queue gets full, double the size and copy elements from `start_` until every element is copied;



Keep `start_` as a pointer to the beginning of the queue, move `start_` to the left when removing element from front.

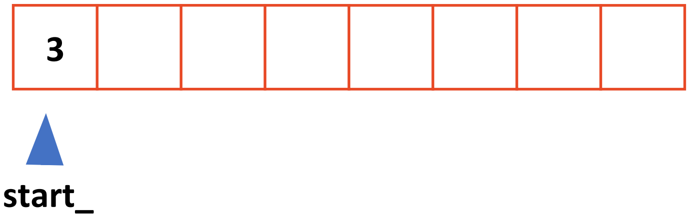
Calculate insertion index: `(start_ + count_)% capacity`




```
Queue&lt;int&gt; q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

start_=0,count_=0 ind=0

```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```



ind =(start_ + count_)%

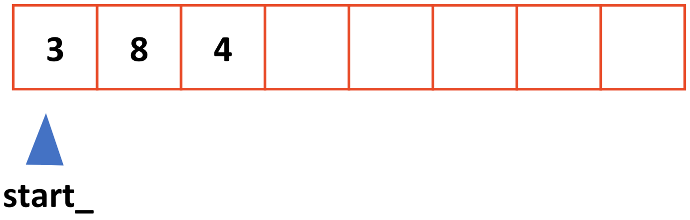

```
start_=0,count_=0 ind=0
start_=0,count_=1 ind=1
```

```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

```
ind =(start_ + count_)%
```

```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

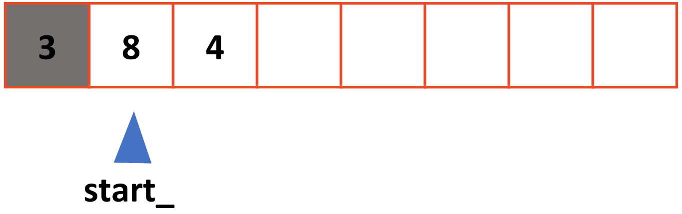
```
start_=0,count_=0 ind=0
start_=0,count_=1 ind=1
start_=0,count_=2 ind=2
```




```
ind =(start_ + count_)%
```

Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);

```
start_=0,count_=0 ind=0
start_=0,count_=1 ind=1
start_=0,count_=2 ind=2
start_=0,count_=3
```




```
start_=1,count_=2
```

```
ind =(start_ + count_)%
```

```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

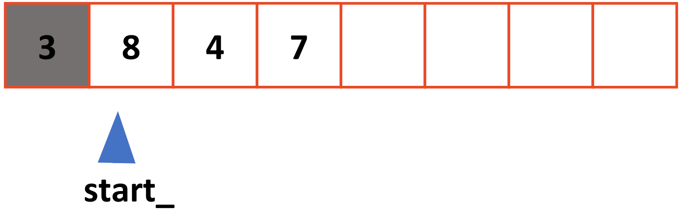
start_=0,count_=0 ind=0

start_=0,count_=1 ind=1

start_=0,count_=2 ind=2

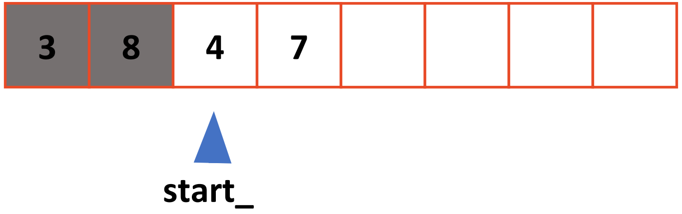
start_=0,count_=3

start_=1,count_=2 ind=3



start_=1,count_=3

ind =(start_ + count_)%

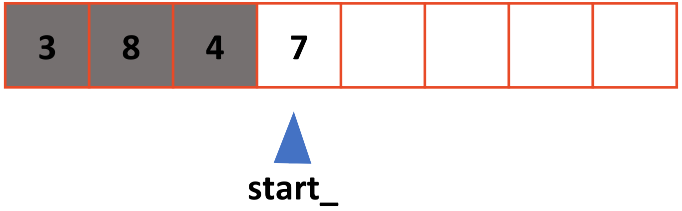


start_=2,count_=2

start_=0,count_=0 ind=0
start_=0,count_=1 ind=1
start_=0,count_=2 ind=2
start_=0,count_=3
start_=1,count_=2 ind=3
start_=1,count_=3

```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

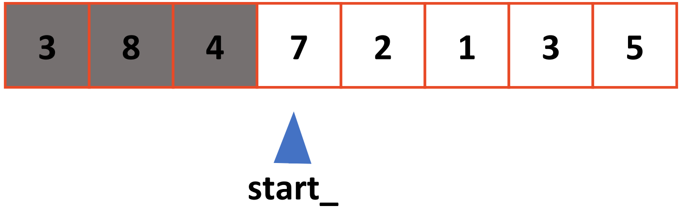
ind =(start_ + count_)%



start_=3,count_=1

		Queue<int> q;
start_=0,count_=0	ind=0	q.enqueue(3);
start_=0,count_=1	ind=1	q.enqueue(8);
start_=0,count_=2	ind=2	q.enqueue(4);
start_=0,count_=3		q.dequeue();
start_=1,count_=2	ind=3	q.enqueue(7);
start_=1,count_=3		q.dequeue();
start_=2,count_=2		q.dequeue();
		q.enqueue(2);
		q.enqueue(1);
		q.enqueue(3);
		q.enqueue(5);
		q.dequeue();
		q.enqueue(9);

ind =(start_ + count_)%



start_=3,count_=5

	Queue<int> q;
start_=0,count_=0 ind=0	q.enqueue(3);
start_=0,count_=1 ind=1	q.enqueue(8);
start_=0,count_=2 ind=2	q.enqueue(4);
start_=0,count_=3	q.dequeue();
start_=1,count_=2 ind=3	q.enqueue(7);
start_=1,count_=3	q.dequeue();
start_=2,count_=2	q.dequeue();
start_=3,count_=1 ind=4	q.enqueue(2);
start_=3,count_=2 ind=5	q.enqueue(1);
start_=3,count_=3 ind=6	q.enqueue(3);
start_=3,count_=4 ind=7	q.enqueue(5);
	q.dequeue();
	q.enqueue(9);

ind =(start_ + count_)%

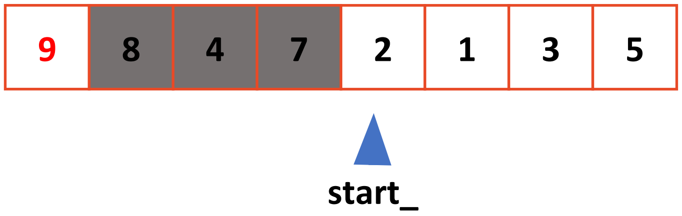
3	8	4	7	2	1	3	5

start_

start_=4,count_=4

		Queue<int> q;
start_=0,count_=0	ind=0	q.enqueue(3);
start_=0,count_=1	ind=1	q.enqueue(8);
start_=0,count_=2	ind=2	q.enqueue(4);
start_=0,count_=3		q.dequeue();
start_=1,count_=2	ind=3	q.enqueue(7);
start_=1,count_=3		q.dequeue();
start_=2,count_=2		q.dequeue();
start_=3,count_=1	ind=4	q.enqueue(2);
start_=3,count_=2	ind=5	q.enqueue(1);
start_=3,count_=3	ind=6	q.enqueue(3);
start_=3,count_=4	ind=7	q.enqueue(5);
start_=3,count_=5		q.dequeue();
		q.enqueue(9);

ind =(start_ + count_)%



start_=4,count_=5

Queue<int> q;

start_=0,count_=0 ind=0 q.enqueue(3);

start_=0,count_=1 ind=1 q.enqueue(8);

start_=0,count_=2 ind=2 q.enqueue(4);

start_=0,count_=3 q.dequeue();

start_=1,count_=2 ind=3 q.enqueue(7);

start_=1,count_=3 q.dequeue();

start_=2,count_=2 q.dequeue();

start_=3,count_=1 ind=4 q.enqueue(2);

start_=3,count_=2 ind=5 q.enqueue(1);

start_=3,count_=3 ind=6 q.enqueue(3);

start_=3,count_=4 ind=7 q.enqueue(5);

start_=3,count_=5 q.dequeue();

start_=4,count_=4 ind=0 $\boxed{\text{q.enqueue(9);}}$

ind =(start_ + count_)%

9	6	11	13	2	1	3	5

start_

start_=4,count_=8

	Queue<int> q;
start_=0,count_=0 ind=0	q.enqueue(3);
start_=0,count_=1 ind=1	q.enqueue(8);
start_=0,count_=2 ind=2	q.enqueue(4);
start_=0,count_=3	q.dequeue();
start_=1,count_=2 ind=3	q.enqueue(7);
start_=1,count_=3	q.dequeue();
start_=2,count_=2	q.dequeue();
start_=3,count_=1 ind=4	q.enqueue(2);
start_=3,count_=2 ind=5	q.enqueue(1);
start_=3,count_=3 ind=6	q.enqueue(3);
start_=3,count_=4 ind=7	q.enqueue(5);
start_=3,count_=5	q.dequeue();
start_=4,count_=4 ind=0	q.enqueue(9);
start_=4,count_=5 ind=1	q.enqueue(6);
start_=3,count_=6 ind=2	q.enqueue(11);
start_=3,count_=7 ind=8	q.enqueue(12);
Resize array	q.enqueue(13);

ind =(start_ + count_)%

Resizing array

Index after resizing : `ind = 9;` **q.enqueue(13);**

When queue gets full, double the size and copy elements from `start_` until every element is copied;

9	**6**	**11**	**13**	**2**	**1**	**3**	**5**

start_

`start_=4,count_=8`

After resizing

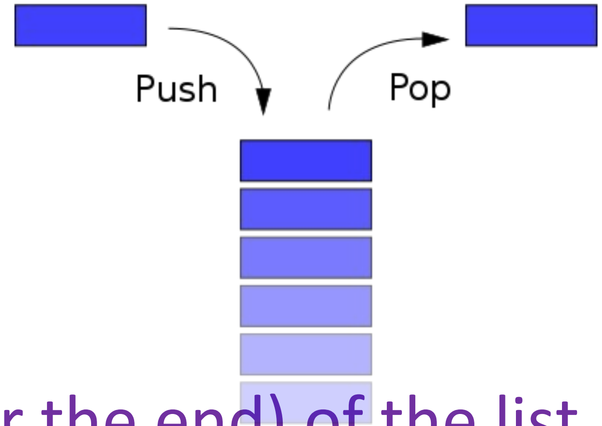
2	**1**	**3**	**5**	**9**	**6**	**11**	**13**								

start_

`start_=0,count_=8, capacity = 16;`

`ind =(start_ + count_)%`

Stack ADT



- [Order]:

Last in First out;

- [Operations]:

push – add element in the beginning (or the end) of the list

pop – delete element from the beginning (or the end) of the list

- [Implementation]:

Removing and adding elements takes O(1) time using List (linked list, array impl.).

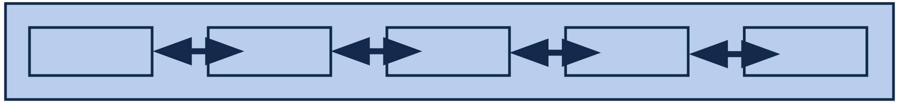
Useful Material:

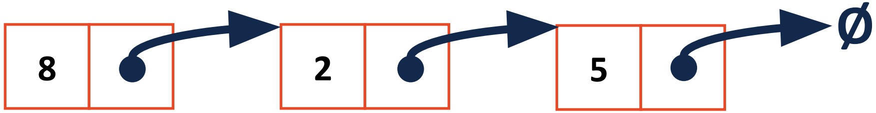
Worksheet
lab_quacks

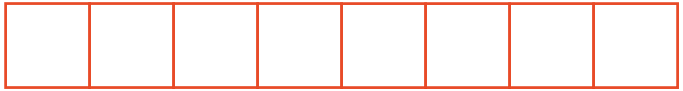
Iterators

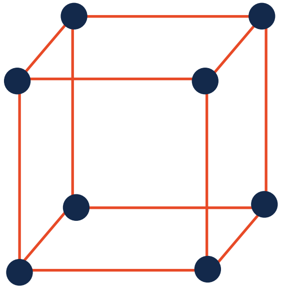
- Iterators
- Operations *, !=, and ++.
- Applications of iterators
- Utility of iterators

Iterators encapsulate access to our data:









Iterators

Every class that implements an iterator has two pieces:

[Implementing Class] in addition has:

1. two member functions:

`::begin()` – Returns an iterator pointing to the first element in the container.

`::end()` – Returns an iterator pointing to the one past-the-end element in the sequence:

Iterators

Every class that implements an iterator has two pieces:

2. [Implementing Class' Iterator]:

- Must have the base class **std::iterator** (pre c++17)
- Must implement

`operator++` – moves to the next data

`operator*` – (dereference op) returns the current data

`operator!=`

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

  return 0;
}
```

`struct` is the same as `class` but all members are public by default

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 mai
  Anima
  std::

  zoo.p
  zoo.p
  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;
}
```

Constructor with default parameters.

```
Animal g("giraffe", "leaves", true)
Animal p("penguin", "fish”)
Animal b("bear");
Animal c(“Fruit");
```

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 mai
  Anima
  std::

  zoo.p
  zoo.p
  zoo.p

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

  return 0;
}
```

Constructor with default parameters.

```
Animal g("giraffe", "leaves", true)
Animal p("penguin", "fish")
Animal b("bear");
Animal c("Fruit");
```

Creates Animal with name = Fruit, default value is used for food ("you"), and big (true)

stlList.cpp

Initialization list

```
#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

```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;
}
```

`vector` implements an array with fast random access and an ability to automatically resize when appending elements 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
  A                                                                  b("bear");
  s



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

  return 0;
}
```

vector<Animal>::iterator type of the `vector<Animal>` iterator `it`

zoo.begin() returns iterator pointing to the first element in the vector

zoo.end() returns iterator pointing to the one past last element of the vector

it++ increments iterator (makes iterator point to next element in the container)

(*it) dereferencing the iterator.

(*it).name accessing `name` field

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

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

C++ will replace this code with the first type of `for` loop .

For each loop: If we have an iterator for the collection we can use for each loop to go over each element In the collection.

General syntax:

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

For Each and Iterators

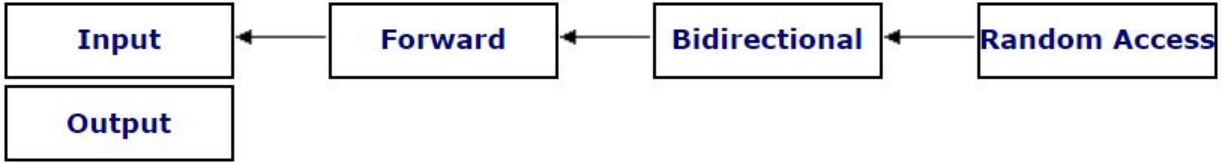
```
for ( const TYPE & variable : collection ) {
   // ...
}
```

As long as container implements iterator we can use foreach loop using same syntax;

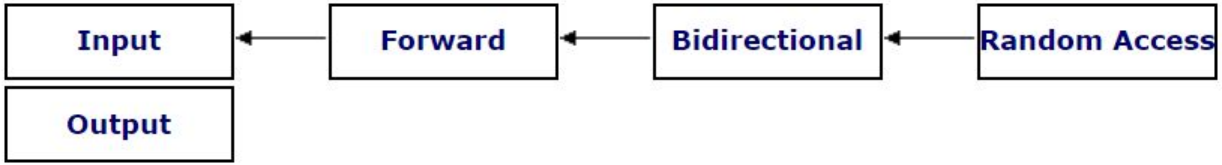
```
std::vector<Animal> zoo;
…
for ( const Animal & animal : zoo ) {
     std::cout << animal.name << " " << animal.food << std::endl;
}
```

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

Iterator categories



Iterator categories



A student clicked on this link. What happened after will shock you:
Useful material:

http://www.cplusplus.com/reference/iterator/

The properties of each iterator category are:

<table>
<tr><th colspan="4">category</th><th>properties</th><th>valid expressions</th></tr>
<tr><td colspan="4" rowspan="2">all categories</td><td>copy-constructible, copy-assignable and destructible</td><td><code>X b(a);</code>
<code>b = a;</code></td></tr>
<tr><td>Can be incremented</td><td><code>++a</code>
<code>a++</code></td></tr>
<tr><td rowspan="10">Random Access</td><td rowspan="6">Bidirectional</td><td rowspan="5">Forward</td><td rowspan="2">Input</td><td>Supports equality/inequality comparisons</td><td><code>a == b</code>
<code>a != b</code></td></tr>
<tr><td>Can be dereferenced as an rvalue</td><td><code>*a</code>
<code>a->m</code></td></tr>
<tr><td>Output</td><td>Can be dereferenced as an lvalue
(only for mutable iterator types)</td><td><code>*a = t</code>
<code>*a++ = t</code></td></tr>
<tr><td rowspan="2"></td><td>default-constructible</td><td><code>X a;</code>
<code>X()</code></td></tr>
<tr><td>Multi-pass: neither dereferencing nor incrementing affects dereferenceability</td><td><code>{ b=a; *a++; *b; }</code></td></tr>
<tr><td colspan="2"></td><td>Can be decremented</td><td><code>--a</code>
<code>a--</code>
<code>*a--</code></td></tr>
<tr><td colspan="3" rowspan="4"></td><td>Supports arithmetic operators + and -</td><td><code>a + n</code>
<code>n + a</code>
<code>a - n</code>
<code>a - b</code></td></tr>
<tr><td>Supports inequality comparisons (<, >, <= and >=) between iterators</td><td><code>a < b</code>
<code>a > b</code>
<code>a <= b</code>
<code>a >= b</code></td></tr>
<tr><td>Supports compound assignment operations += and -=</td><td><code>a += n</code>
<code>a -= n</code></td></tr>
<tr><td>Supports offset dereference operator ([])</td><td><code>a[n]</code></td></tr>
</table>

Trees

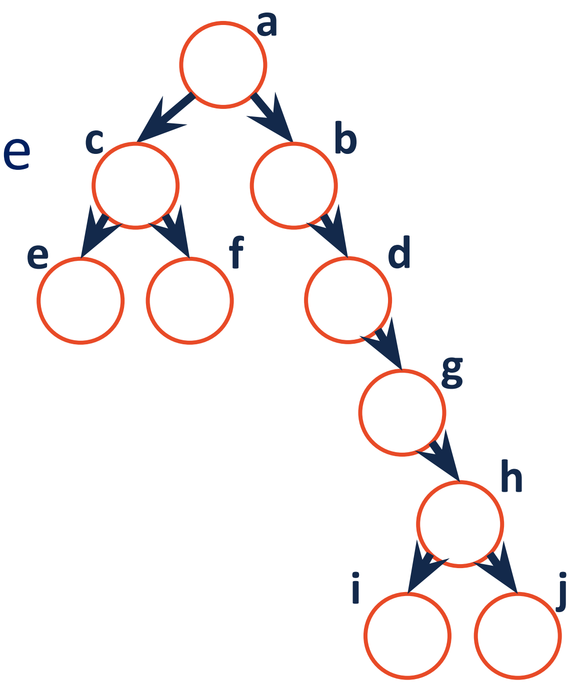
- Basic tree terminology (CS 173)
- Tree Property: Binary
- Tree Property: Height
- Tree Property: Full
- Tree Property: Perfect
- Tree Property: Complete (as defined in data structures)
- Tree Property: NULL pointers in a BST, including proof
- Traversal: Pre-Order, In-Order, and Post-Order
- Traversal: Level-Order
- Search Strategy: BFS, DFS

binary trees:

- Each node in a binary tree contains only **two or fewer children** – where one is the “left child” and one is the “right child”:

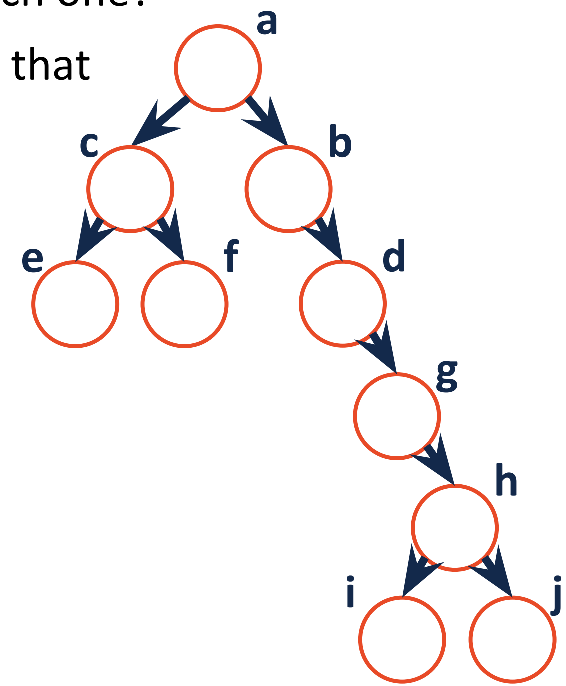
- A binary tree is **rooted** – every node can be reached via a path from the root

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



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?
- Make an "word" containing the names of the vertices that have a **parent** but no **sibling**.
- 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.



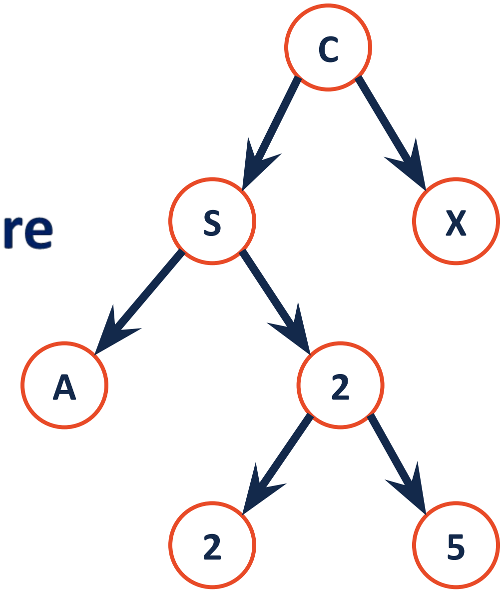
Binary Tree – Defined

- A *binary tree* T is either:

  - $T = \{r, T_l, T_R\}$ where T_l and T_R are binary trees

    OR

  - $T = \{\,\} = \emptyset$



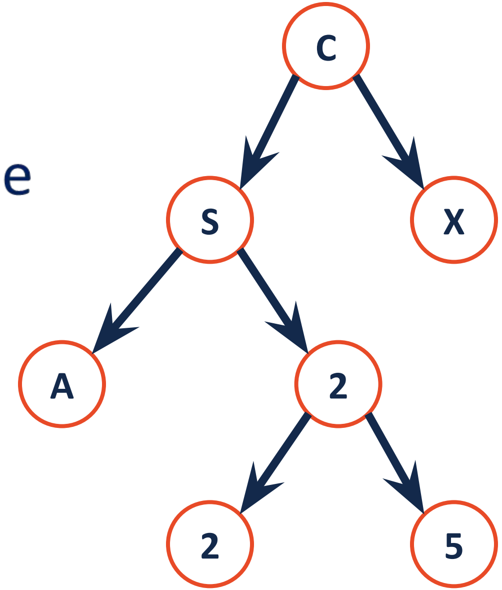
Tree Property: height

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

Given a binary tree T:

$$height(T) = max(height(T\_l\ ), height(T\_R) + 1$$

$$height(T\{\ \}) = \textit{-1}$$

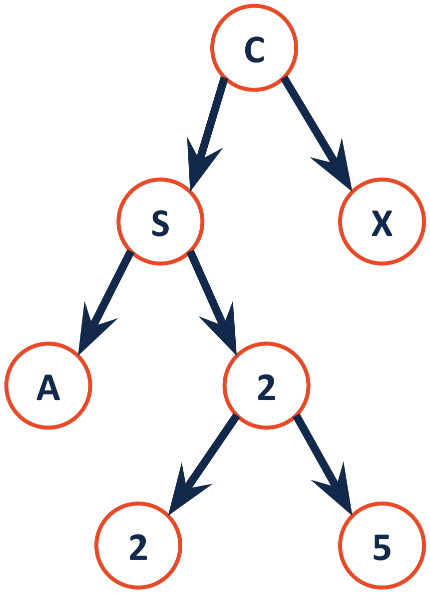


Tree Property: full

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

  1. $F\{\}$

  2. $\boldsymbol{F = \{r, T_l, T_R\}}$ **where** $\boldsymbol{T_l}$ **and** $\boldsymbol{T_R}$ **both have zero or two children.**



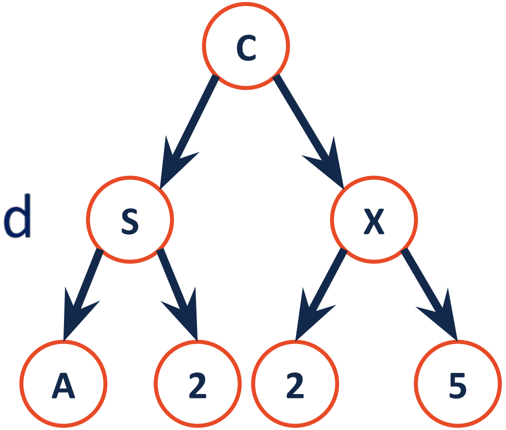
Tree Property: perfect

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

  Let P_h be an perfect tree of height h, and

  1. $P_{-1} = \{\ \}$

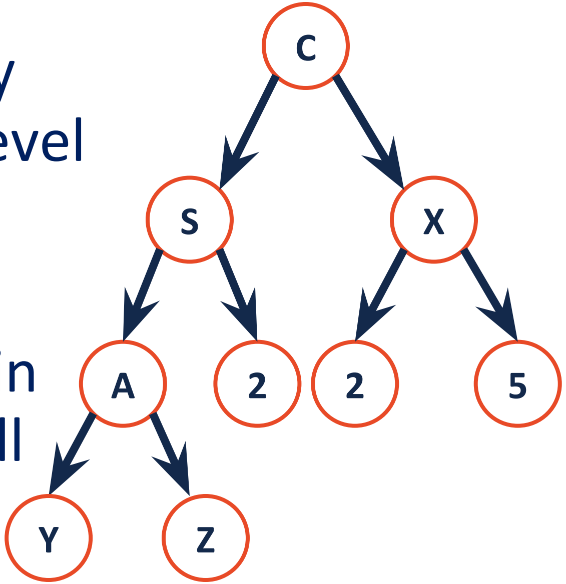
  2. $P_h = \{\mathrm{r}, \mathrm{P}_{\mathrm{h}-1}, \mathrm{P}_{\mathrm{h}-1}\}, \mathrm{h} \geq 0$



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

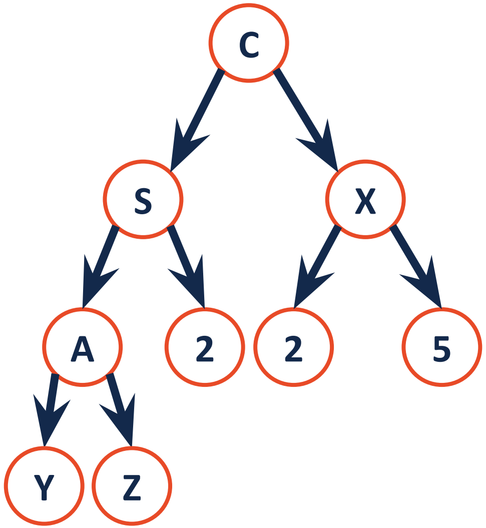
- A **complete** tree ***C*** of height **h, C_h**:

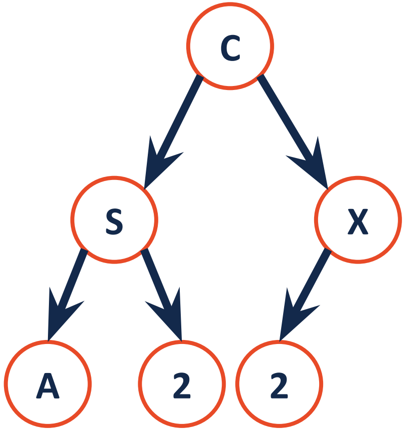
  1. **C_{-1} = {}**
  2. **C_h** *(where h>0)* **= {r, T_L, T_R}** and either:

     T_L is C_{h-1} and T_R is P_{h-2}

     OR

     T_L is P_{h-1} and T_R is C_{h-1}

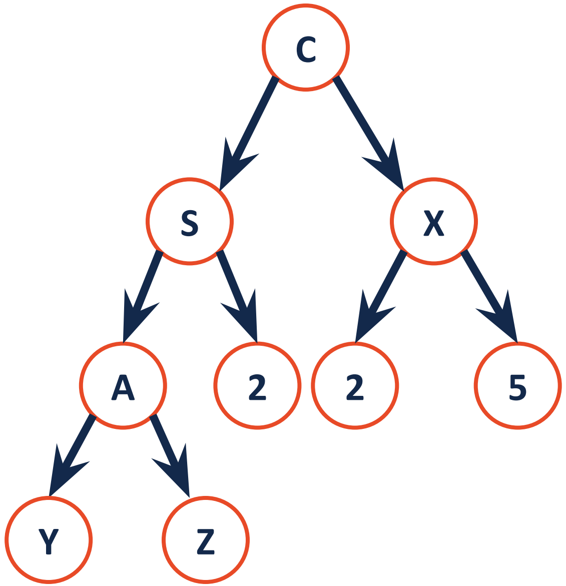




Tree Property: complete

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

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

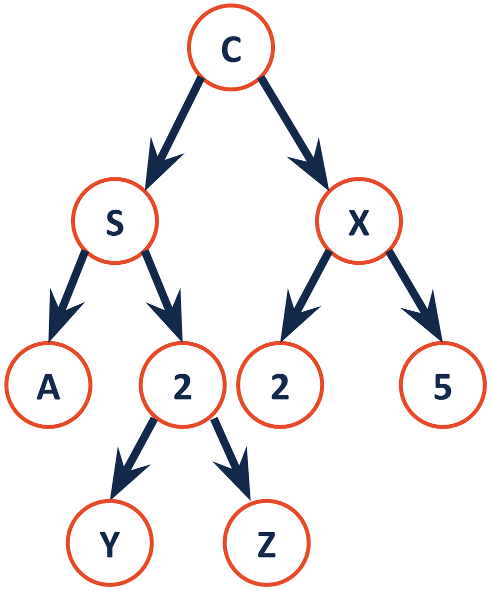


Tree Property: complete

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

NO

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



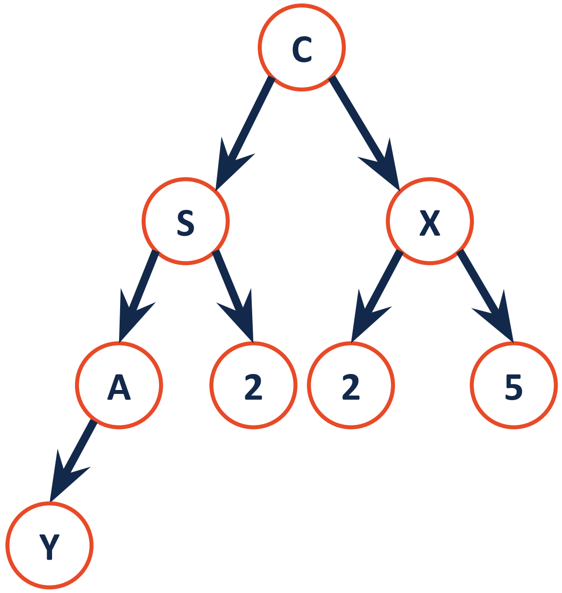
Tree Property: complete

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

NO

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

NO



Tree ADT

- Functionalities
  - Insert
  - Remove
  - Traverse
- A binary tree is like a fancy linked list
  - TreeNode

```
#pragma once

template <typename T>
class BinaryTree {
     public:

          /* … */

     private:
          class TreeNode {
               TreeNode * left;  // pointer to the left child
               TreeNode * right; // pointer to the right child
               T & data;
               TreeNode(T & t)  :
                  data(t), left(NULL), right(NULL) {};
                  // constructor (initialization list)
          }
          TreeNode * root_;
          // root of the tree: similar to head in linked list
}

…
```

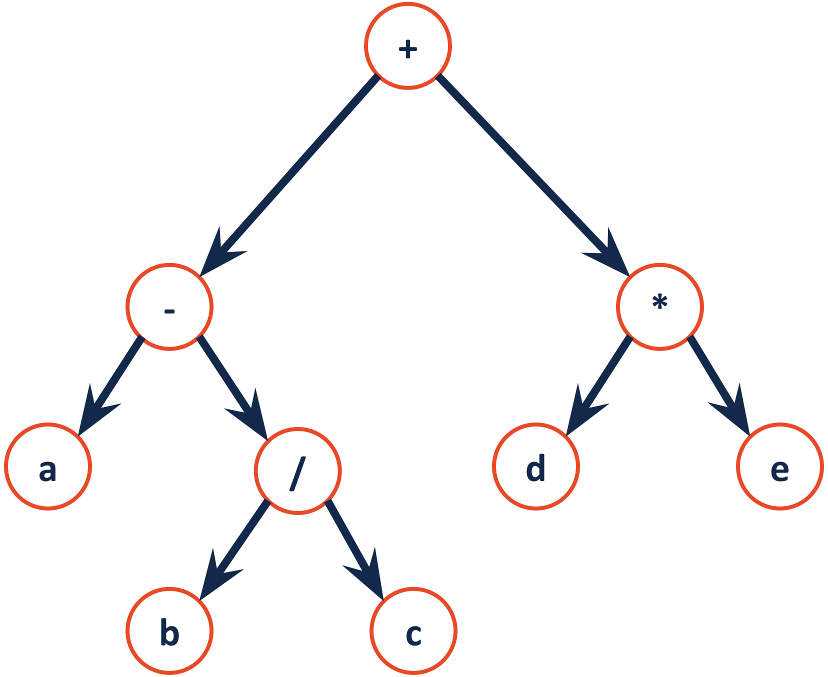
NULL pointers in a BST, including proof

Theorem:

A binary tree with **n** data items has **n+1** null pointers.

Notes for lecture: Tree Traversal -
https://docs.google.com/document/d/1GHBwSXhoor-D-UWqXIv1OEmm3iLmoHGWHmH4wjopUcQ/edit#heading=h.h805liat687e

Traversals



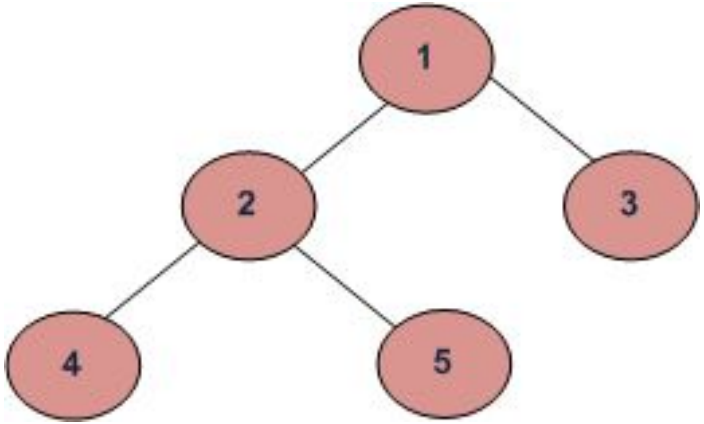
Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

(a) Inorder (Left, root, Right) : 4 2 5 1 3

(b) Preorder (root, Left, Right) : 1 2 4 5 3

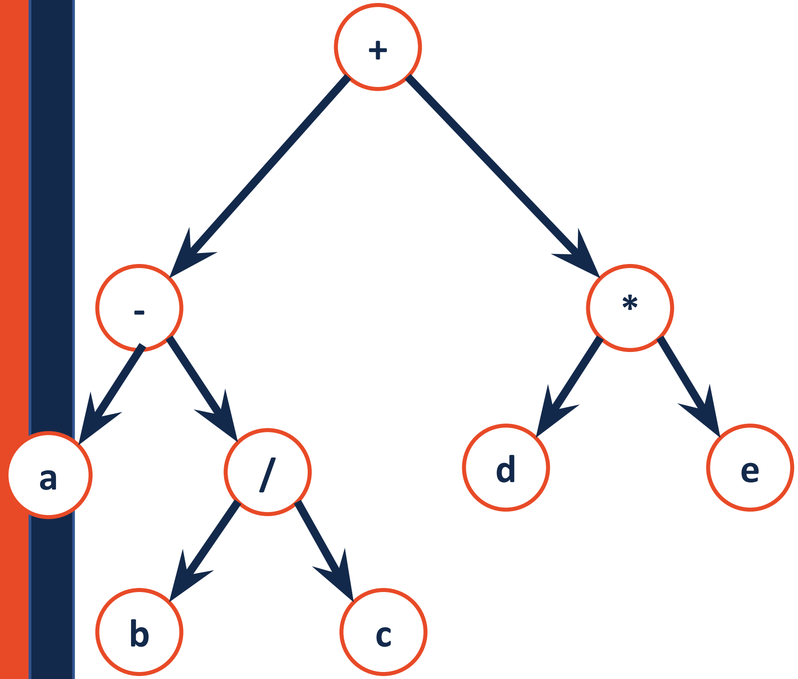
(c) Postorder (Left, Right, Root) : 4 5 2 3 1



Tree traversals refer to the process of visiting each node in the tree data structure exactly once.

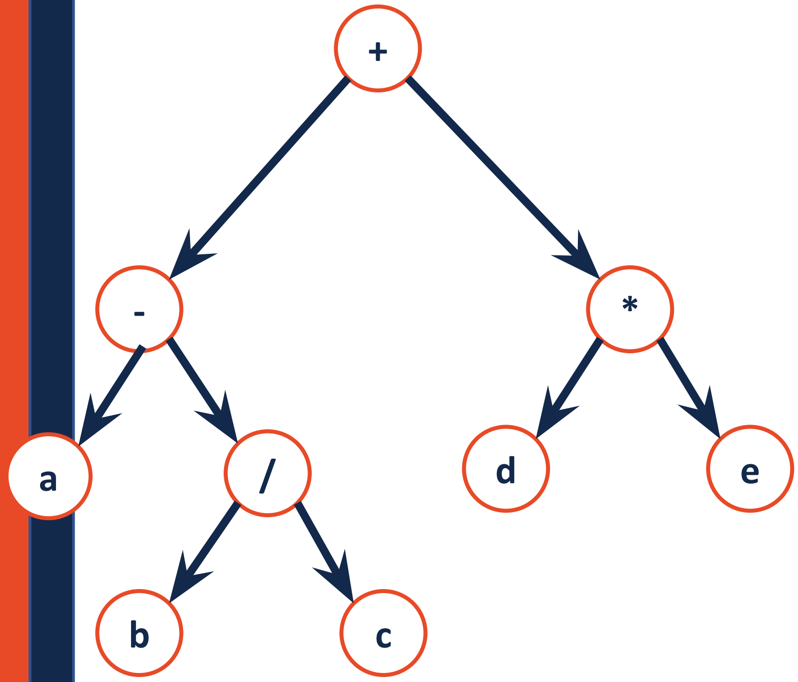
https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postord

Traversals



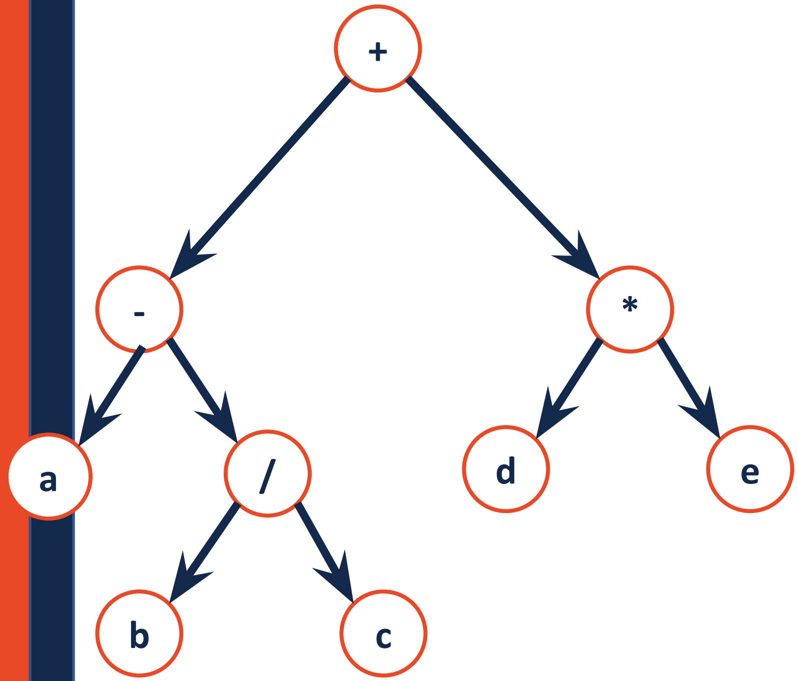

```
template<class T>
void BinaryTree<T>::___Order(TreeNode * cur) {
   if (cur != NULL) {

      ______________________________;

      ___Order(cur->left);

      ______________________________;

      ___Order(cur->right);

      ______________________________;

   }
}
```

Traversals



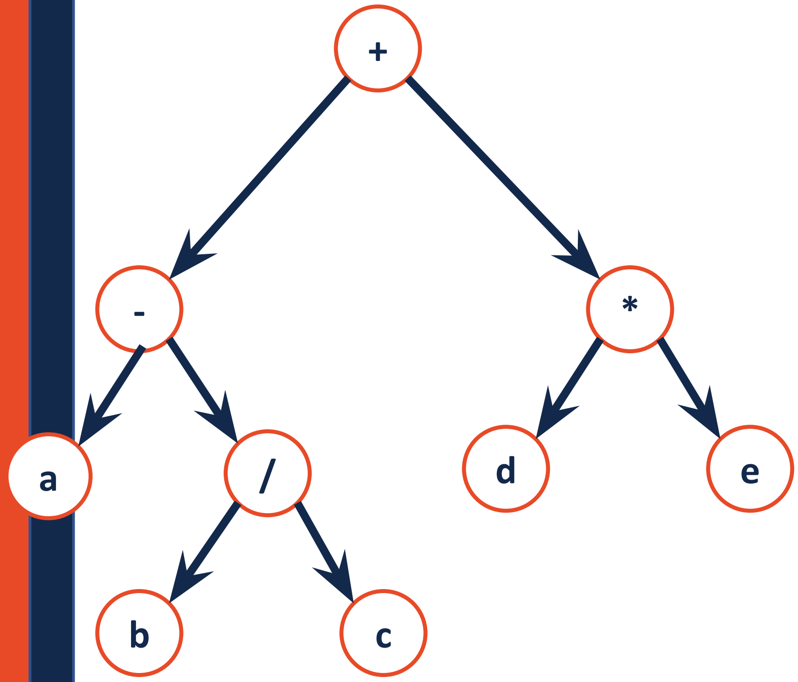

```
template<class T>
void BinaryTree<T>::PreOrder(TreeNode * cur) {
   if (cur != NULL) {

         Yell(cur->data);
         PreOrder(cur->left);

  //     _______________________;

         PreOrder(cur->right);

  //    _______________________;

  }
}
```

Traversals



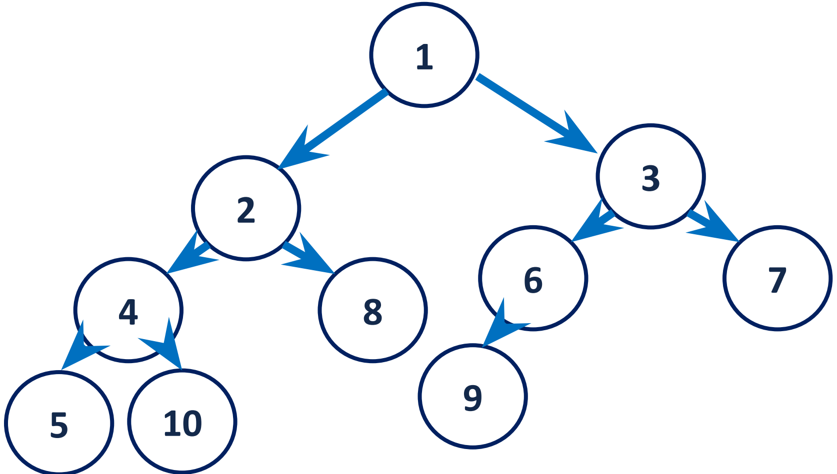

```
template<class T>
void BinaryTree<T>::InOrder(TreeNode * cur) {
   if (cur != NULL) {

  //      ______________________;

        InOrder(cur->left);

        Yell(cur->data);

        InOrder(cur->right);

   //     ______________________;

  }
}
```

Traversals

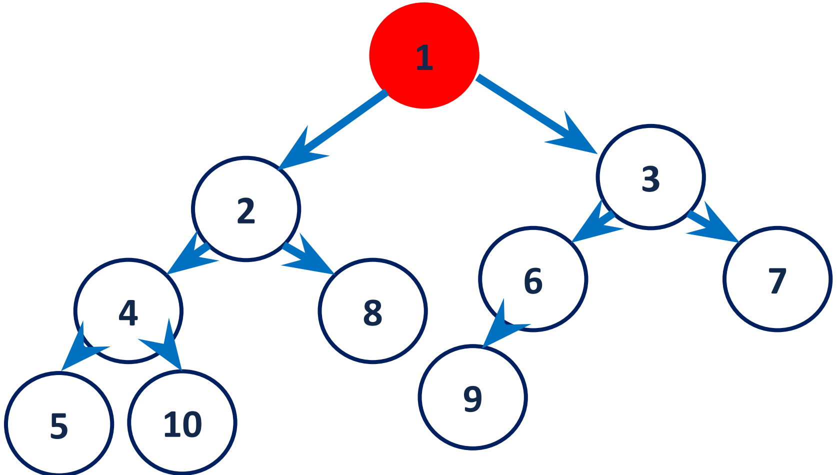



```
template<class T>
void BinaryTree<T>::PostOrder(TreeNode * cur) {
   if (cur != NULL) {

  //      __________________________;

        PostOrder(cur->left);

  //      __________________________;

        PostOrder(cur->right);

        Yell(cur->data);

  }
}
```

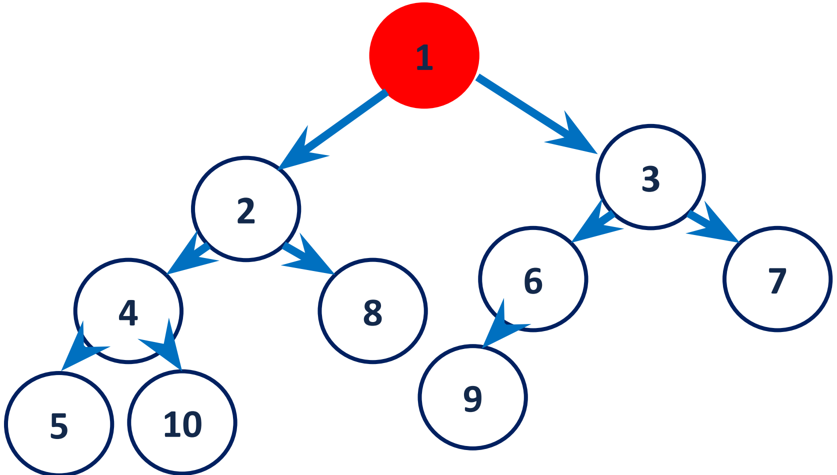
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```



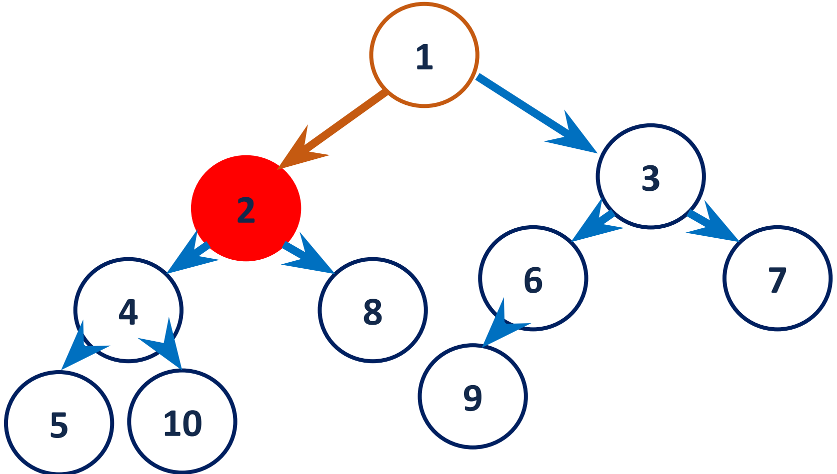
```
void printInOrder(const Node* subRoot) const {
  if(subRoot == NULL) {
  return;
  }
  printInOrder(subRoot->left);
  cout << subRoot->value << " ";
  printInOrder(subRoot->right);
}
```



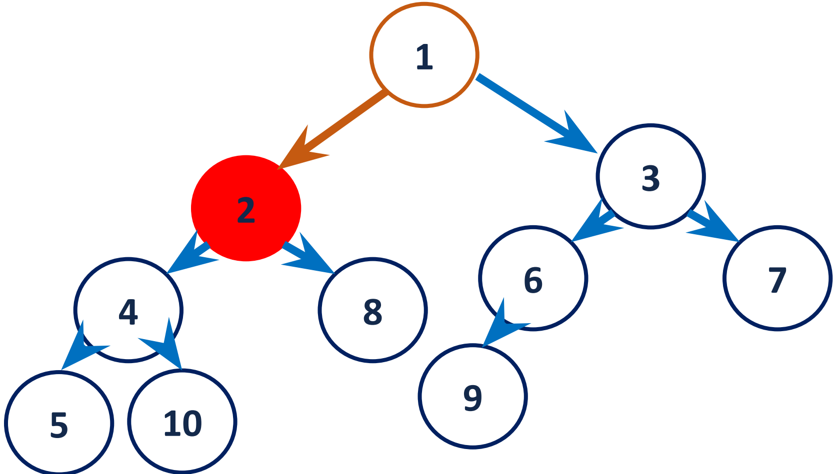
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << “ ”;
 printInOrder(subRoot->right);
}
```



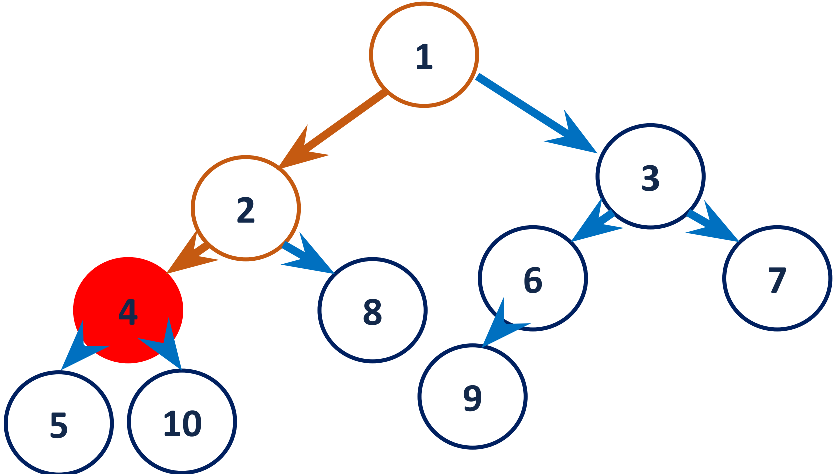
```
void printInOrder(const Node* subRoot) const {
  if(subRoot == NULL) {
  return;
  }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```



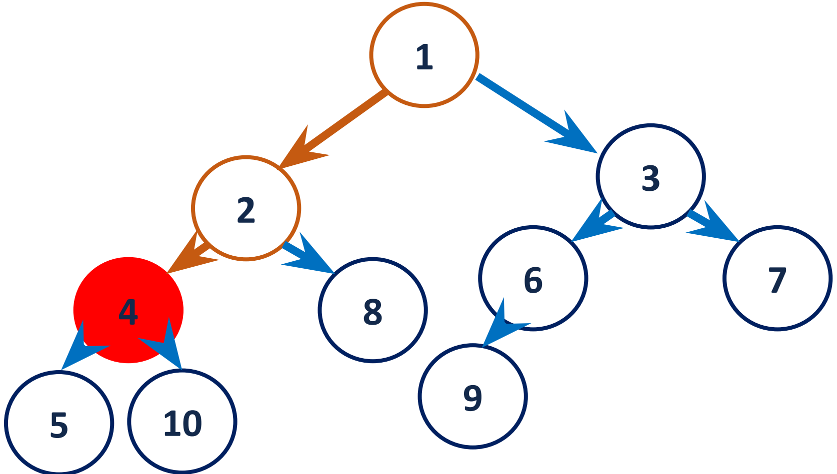
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << “ “;
 printInOrder(subRoot->right);
}
```



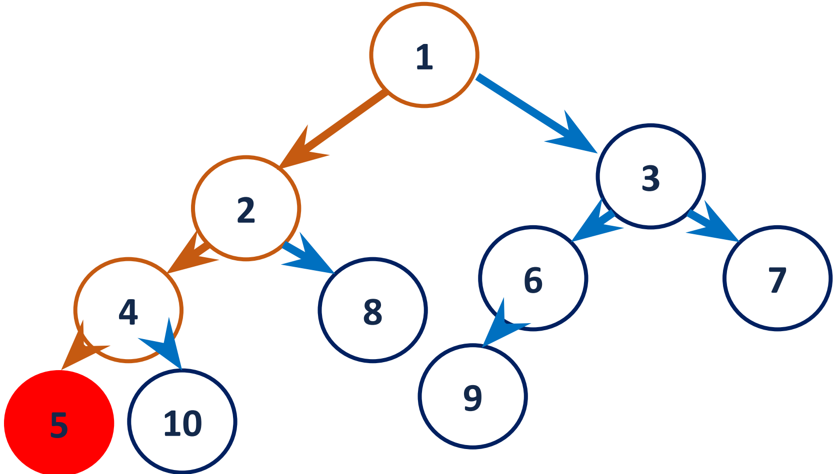
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```



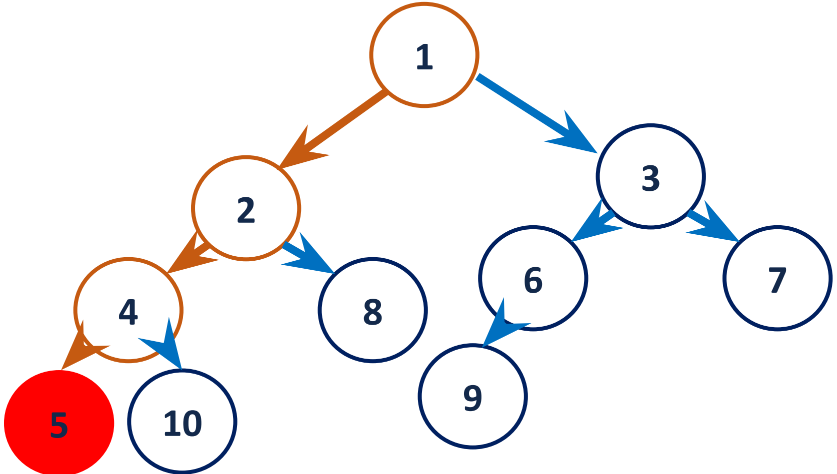
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```



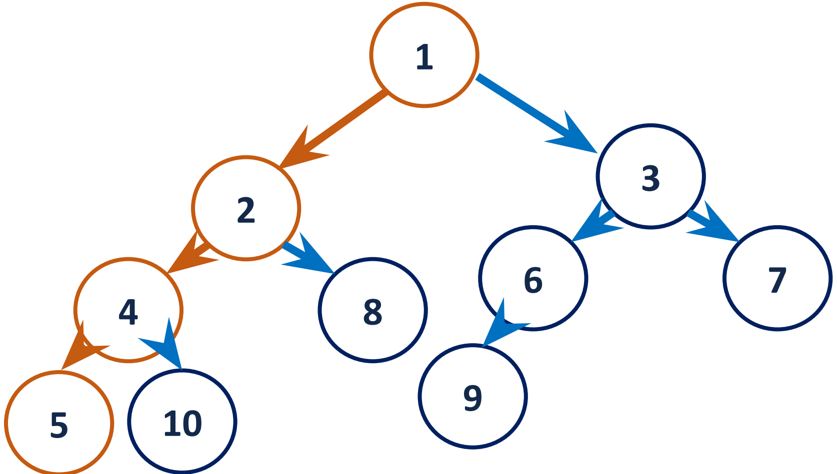
```
void printInOrder(const Node* subRoot) const {
  if(subRoot == NULL) {
  return;
  }
  printInOrder(subRoot->left);
  cout << subRoot->value << “ “;
  printInOrder(subRoot->right);
}
```



```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```

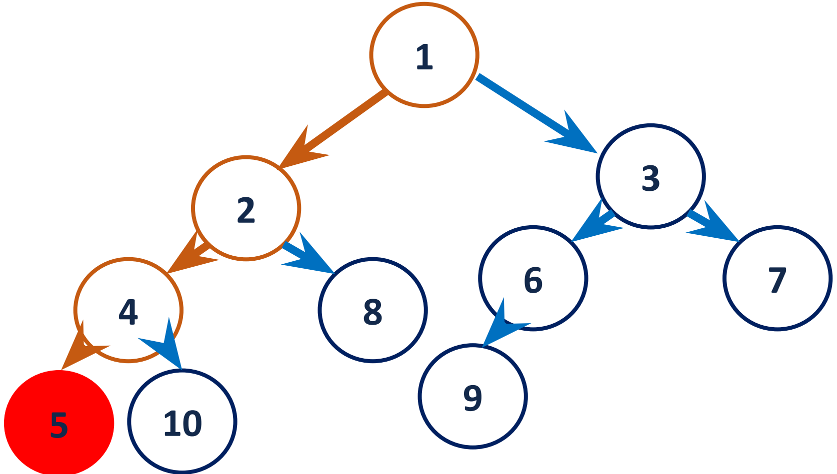


```
void printInOrder(5->left(=NULL)) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```



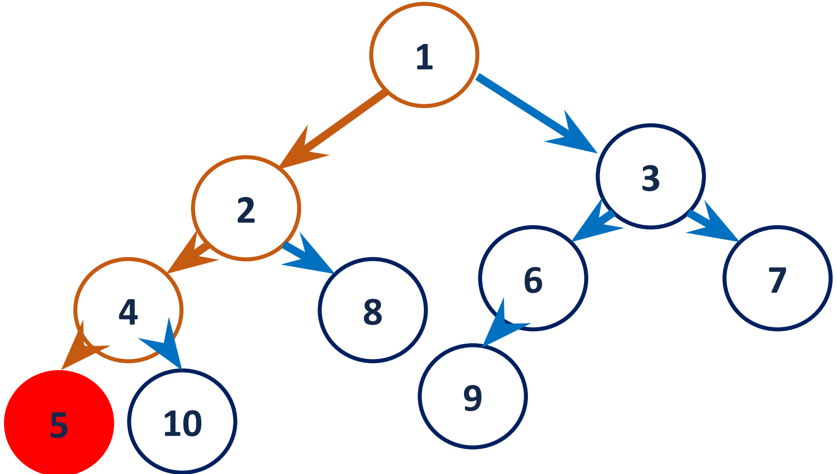
```
void printInOrder(const Node* subRoot) const {
  if(subRoot == NULL) {
  return;
  }
  printInOrder(subRoot->left);
  cout << subRoot->value << “ “;
  printInOrder(subRoot->right);
}
```

5



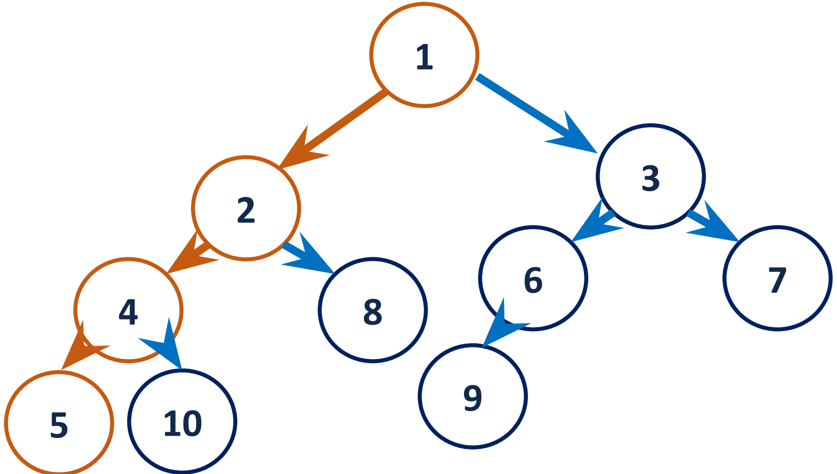
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << “ “;
 printInOrder(subRoot->right);
}
```

5



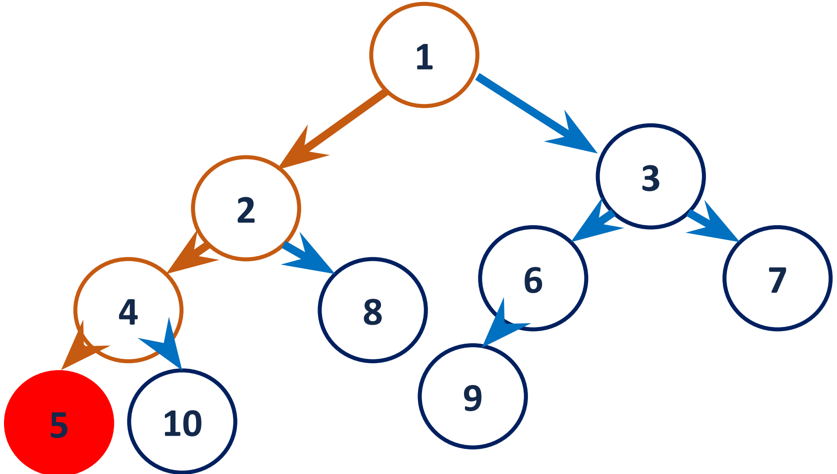
```
void printInOrder(5->right(=NULL)) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << “ “;
 printInOrder(subRoot->right);
}
```

5



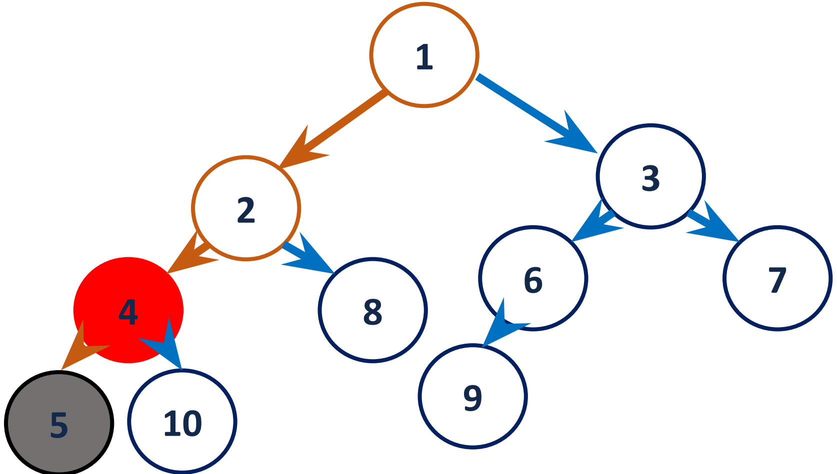
```
void printInOrder(const Node* subRoot) const {
  if(subRoot == NULL) {
  return;
  }
  printInOrder(subRoot->left);
  cout << subRoot->value << " ";
  printInOrder(subRoot->right);
}
```

5



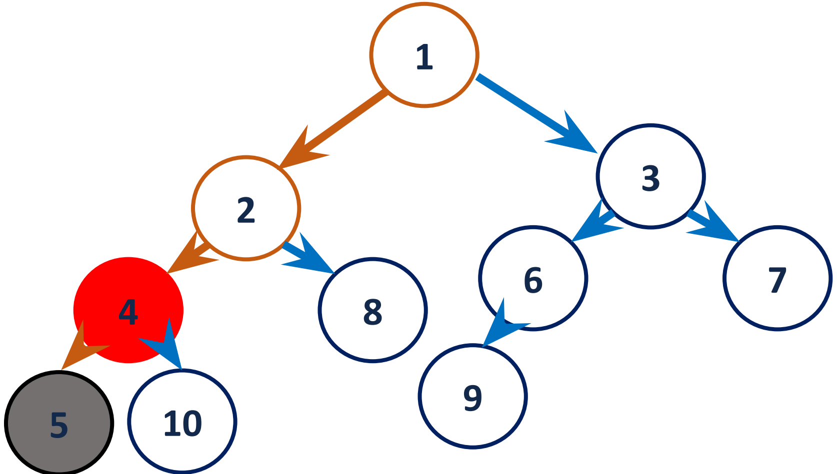
```
void printInOrder(const Node* subRoot) const {
  if(subRoot == NULL) {
  return;
  }
  printInOrder(subRoot->left);
  cout << subRoot->value << " ";
  printInOrder(subRoot->right);
}
```

5	4



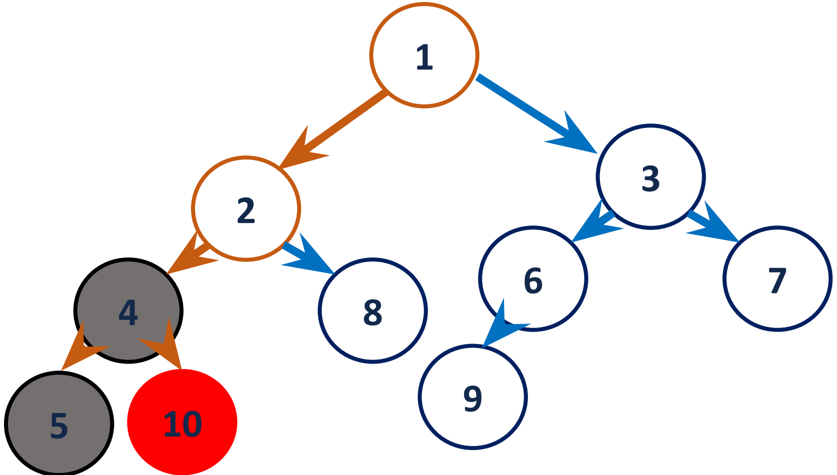
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```

5	4



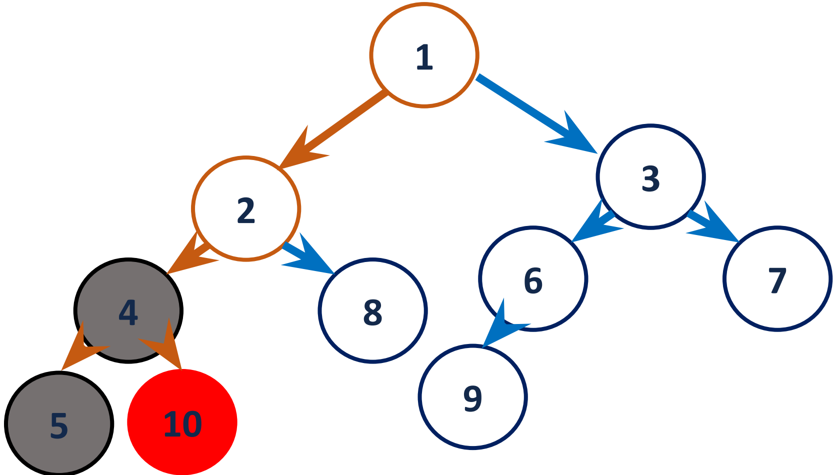
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << “ “;
 printInOrder(subRoot->right);
}
```

5	4



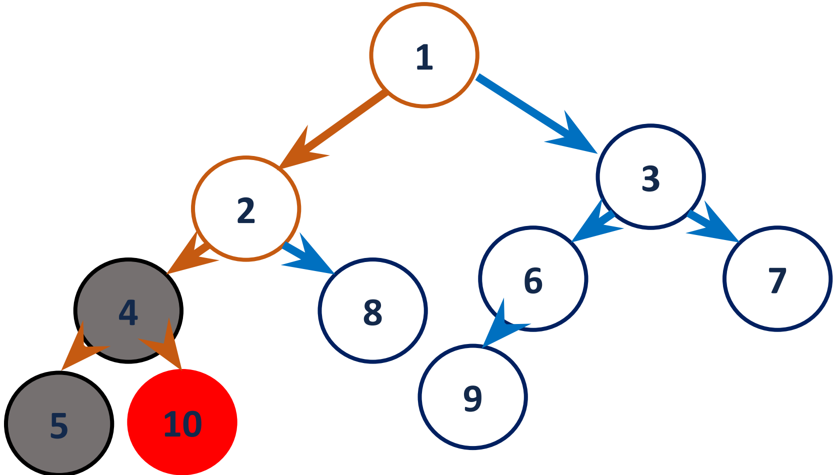
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```

5	4	10



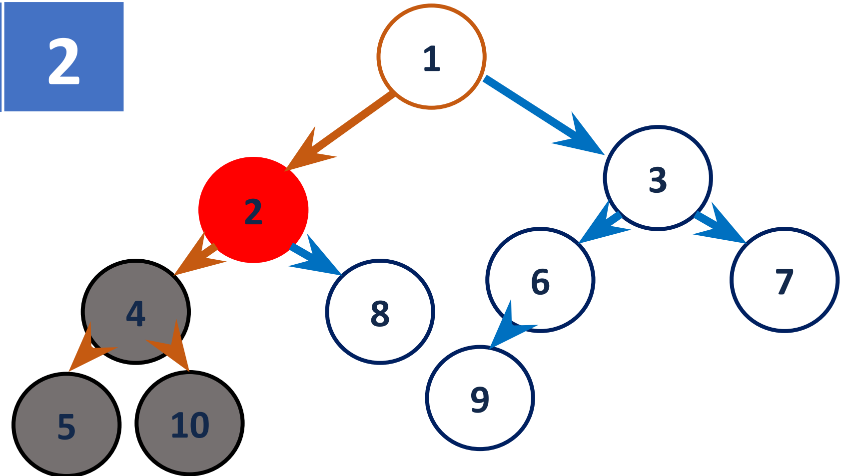
```
void printInOrder(const Node* subRoot) const {
  if(subRoot == NULL) {
  return;
  }
  printInOrder(subRoot->left);
  cout << subRoot->value << " ";
  printInOrder(subRoot->right);
}
```

5	4	10



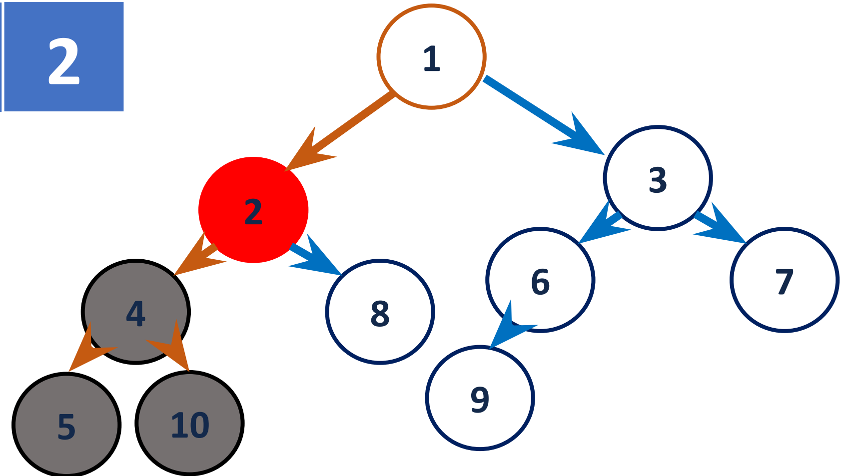
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << “ “;
 printInOrder(subRoot->right);
}
```

5	4	10	2



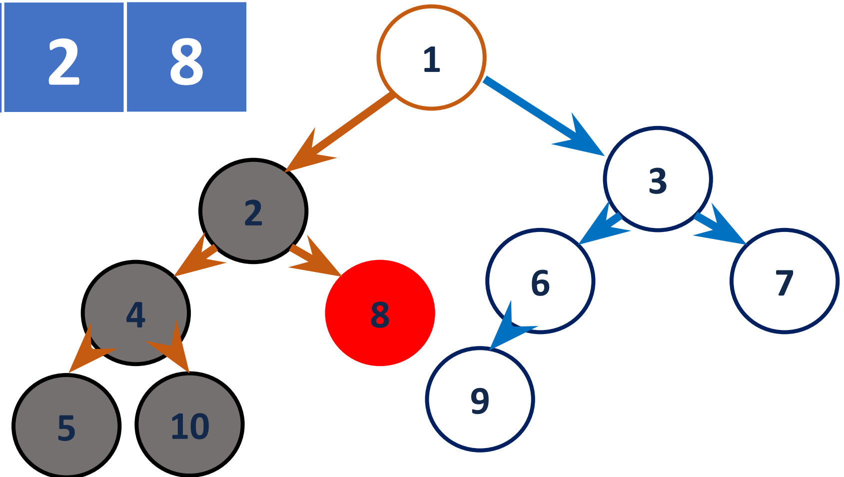
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```

5	4	10	2



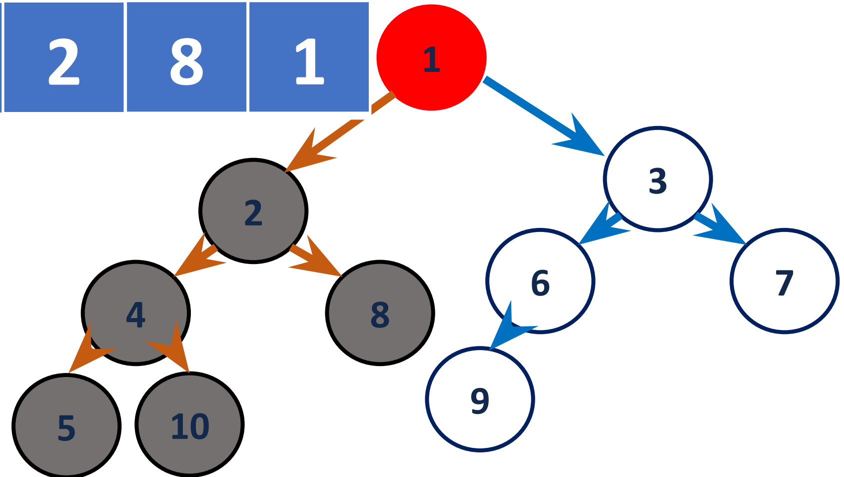
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << " ";
 printInOrder(subRoot->right);
}
```

5	4	10	2	8



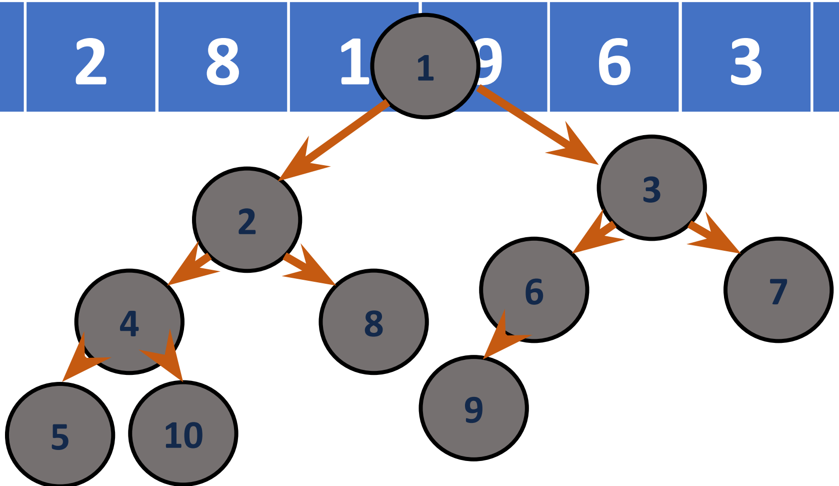
```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << “ “;
 printInOrder(subRoot->right);
}
```

5	4	10	2	8	1

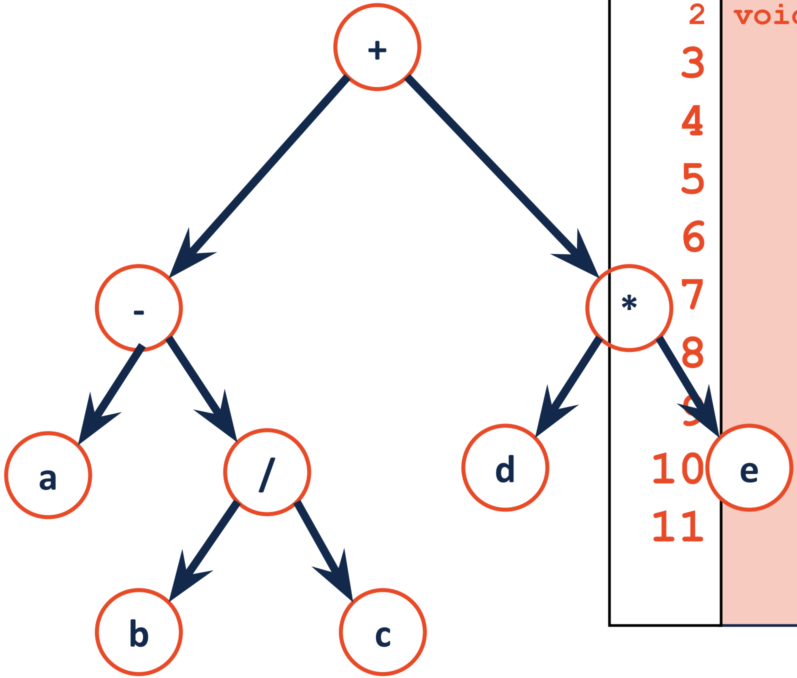


```
void printInOrder(const Node* subRoot) const {
 if(subRoot == NULL) {
 return;
 }
 printInOrder(subRoot->left);
 cout << subRoot->value << “ “;
 printInOrder(subRoot->right);
}
```

5	4	10	2	8	1	9	6	3	7



A Different Type of Traversal




```
template<class T>
void BinaryTree<T>::levelOrder(TreeNode * root) {
   queue<treeNode *> Q;
   Q.enqueue(root);
   While (!Q.isEmpty())
      treeNode * t = q.dequeue();
      if (t!=null)
         yell(t->data);
         q.enqueue(t->left);
         q.enqueue(t->right);
```

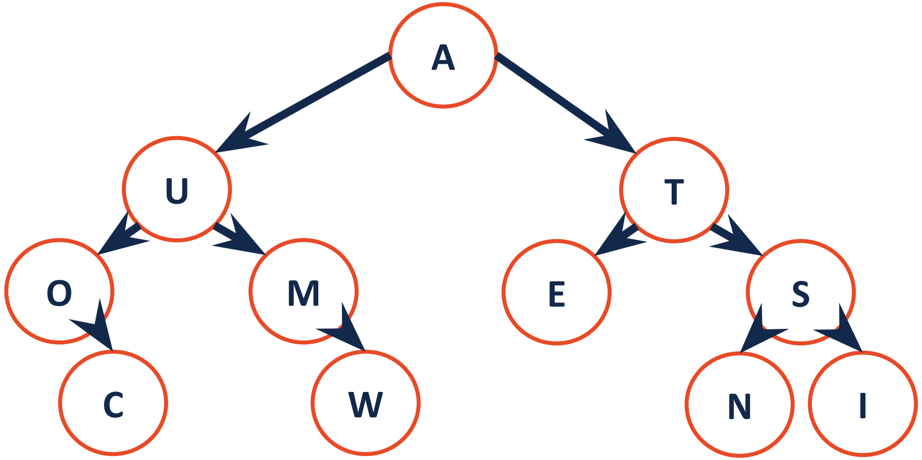
BFS

Start at the tree root, and explore all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

DFS

Start at the root node and explore as far as possible along each branch before backtracking.

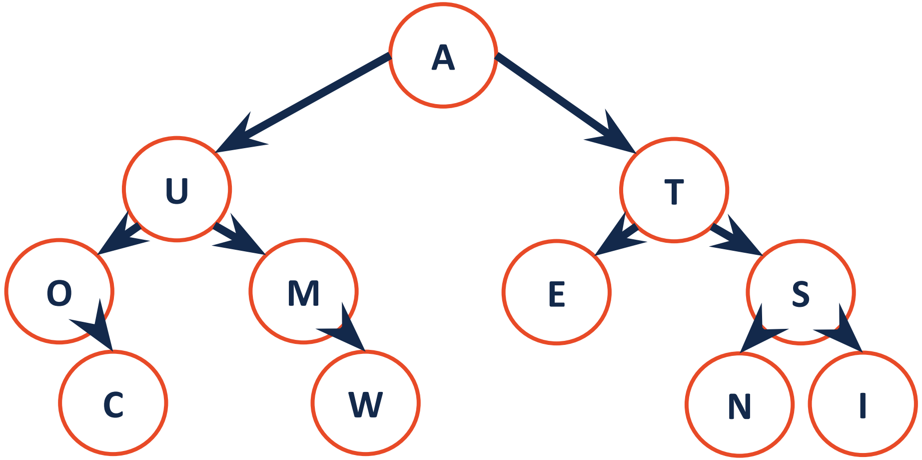
Search – Running time



Worst case:

Best case:

Search – Running time



Worst case: **O(n)**

Worst worst - Data is not in the tree =(

Best case: **Data is in the root -> O(1)**

Binary Search Tree

- Difference between a “Tree” and a “BST”
- Operation find, including running times in terms of h and n
- Operation insert, including running times in terms of h and n
- Operation delete, including running times in terms of h and n
- Strategy for a 2-child delete
- BST Property: Min/max nodes in a tree of a given h, and properties
- BST Property: Height balance factor, b

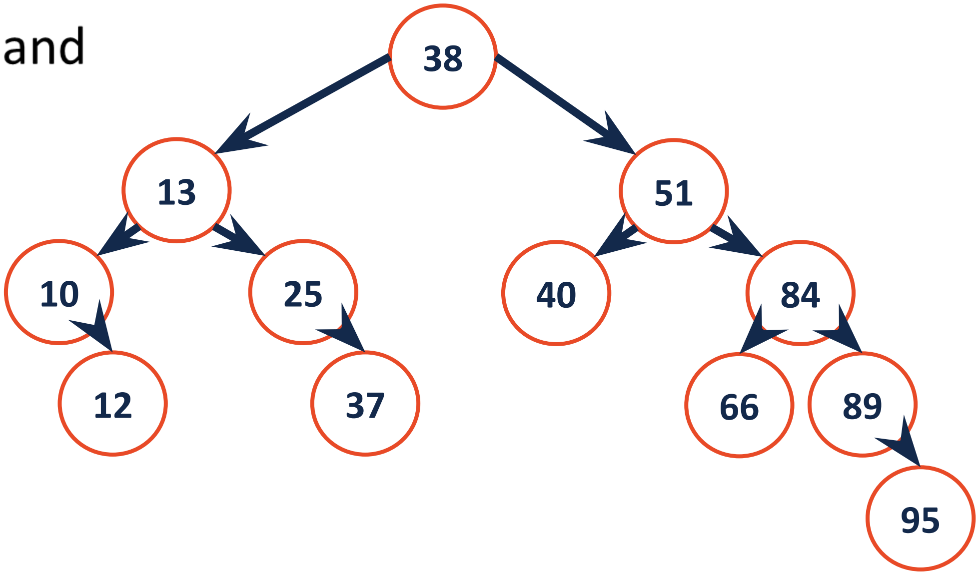
Binary Search Tree (BST)

- A **BST** is a binary tree **T** such that:

$$T = \{\} \quad OR$$

$$T = \{r, T_L, T_R\} \; and$$

$$data(T_L) < r < data\,(T_R) \text{ and}$$
$$T_L \; and \; T_R \; are \; BSTs$$

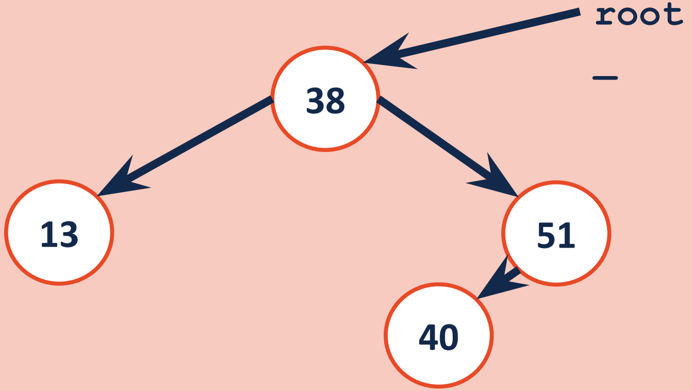


BST.h

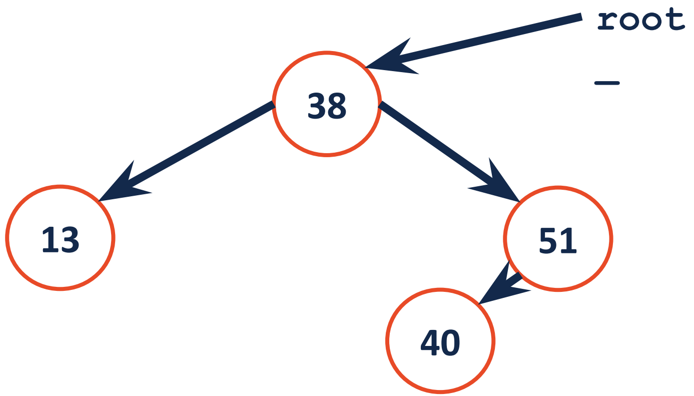
```
#pragma once

template <typename K, typename V>
class BST {
  public:
    BST();
     void insert(const K key, V value);
     V remove(const K & key);
     V find(const K & key) const;
     TreeIterator traverse() const;

  private:
    struct TreeNode {
       TreeNode *left, *right;
       K & key;
       V & value;
       TreeNode(K & k, V & v) : key(k), value(v), left(NULL),
          right(NULL) { }
    };

    TreeNode *head_;
};
```

```
template<typename K, typename V>
V & BST<K,V>::find(const K & key) const {
  TreeNode *& cur = _find(root_, key);
  if (cur != NULL)
    return cur->value;
  else
    throw exception or return a special value;
}
```

```
// helper function
// The return type is a reference; the advantage: even if the
//  key doesn't exist, we return a NULL, the pointer is exactly
//  where the key should be at if we insert it

template<typename K, typename V>
TreeNode *& BST<K,V>::_find(TreeNode *& root, const K & key) const {
  if (root == NULL || key == root->key) {
    return root;
   // when root is NULL, this root pointer is exactly where the key
   //should be
  }
  if (key < root->key) {
    return _find(root->left, key);
  }
  if (key > root->key) {
    return _find(root->right, key);
  }
}
```

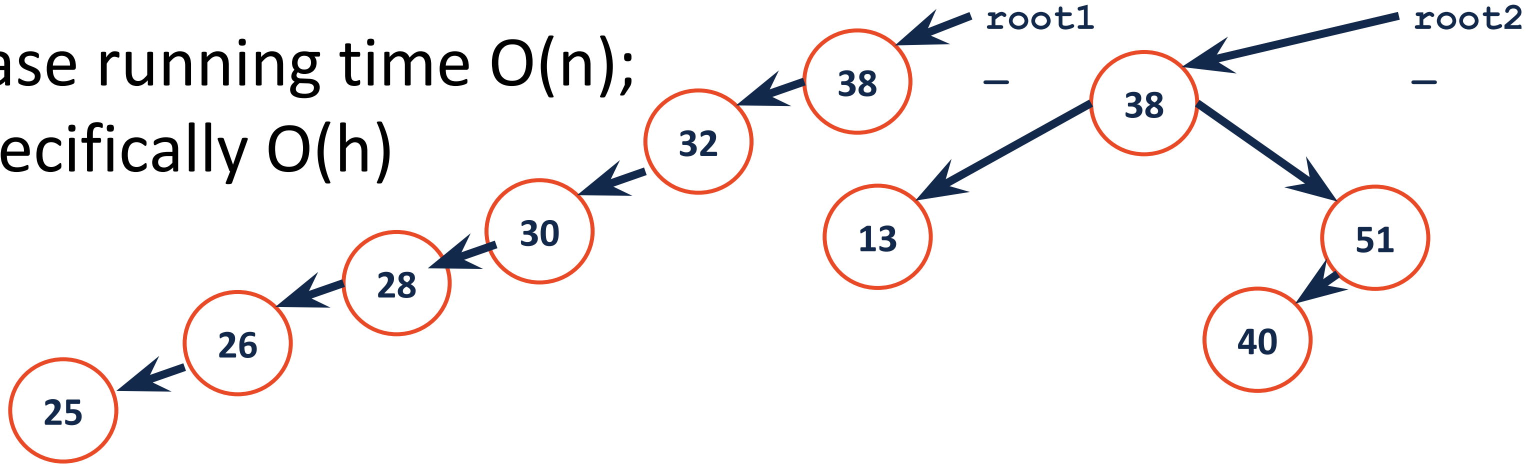


```
// This is easy since _find returns the reference of pointer
template<typename K, typename V>
void BST<K,V>::insert(const K & key, const V & value) {
  TreeNode *& cur = _find(root_, key);
  cur = new TreeNode(key, value);
}
```

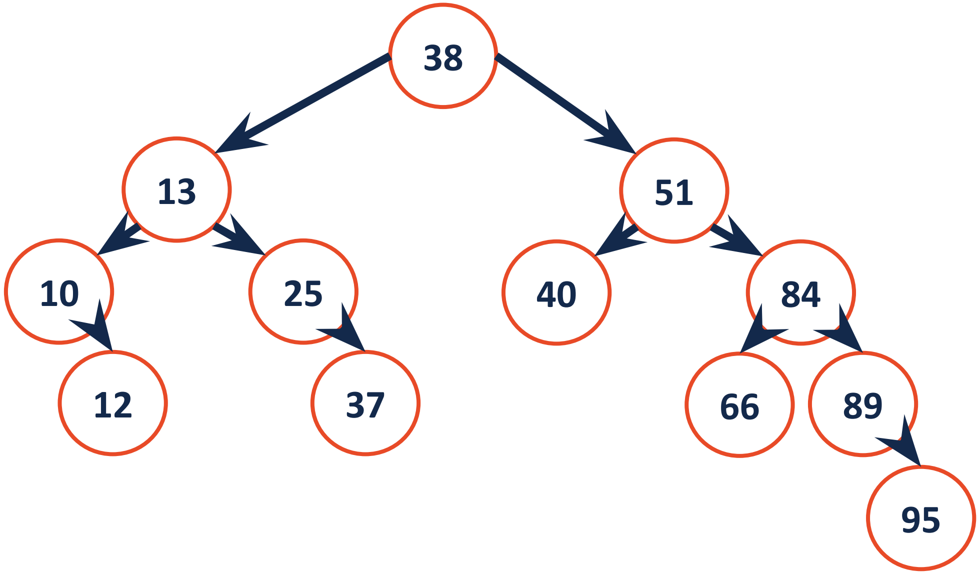


```
// This is easy since _find returns the reference of pointer
template<typename K, typename V>
void BST<K,V>::insert(const K & key, const V & value) {
  TreeNode *& cur = _find(root_, key);
  cur = new TreeNode(key, value);
}
```

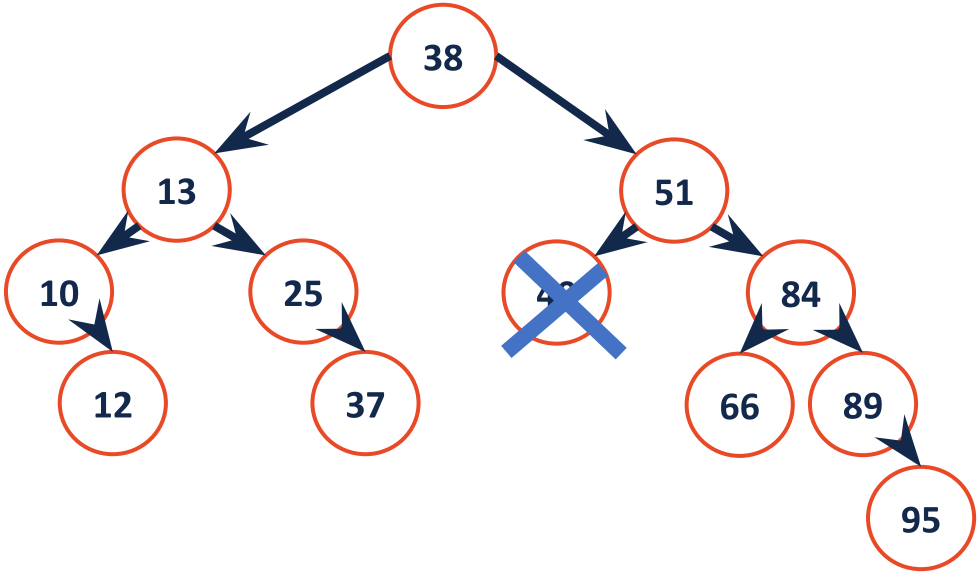
Worst case running time O(n);
More specifically O(h)



Removing element from BST

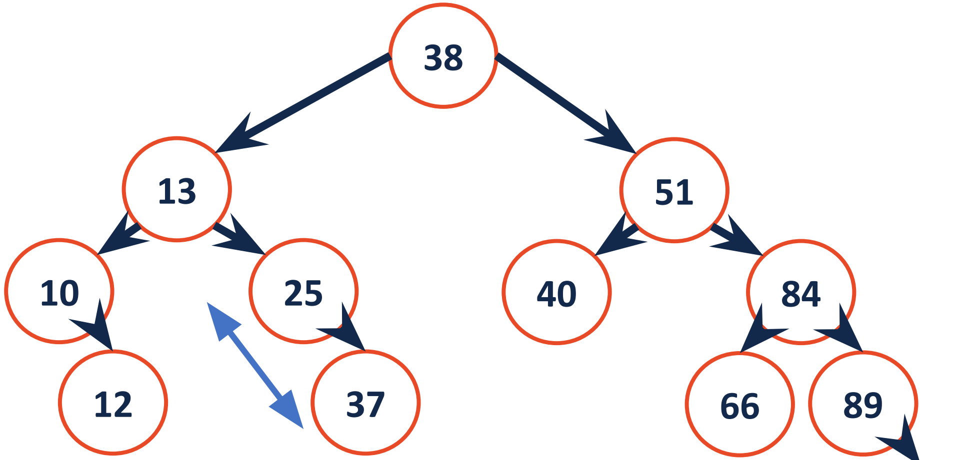


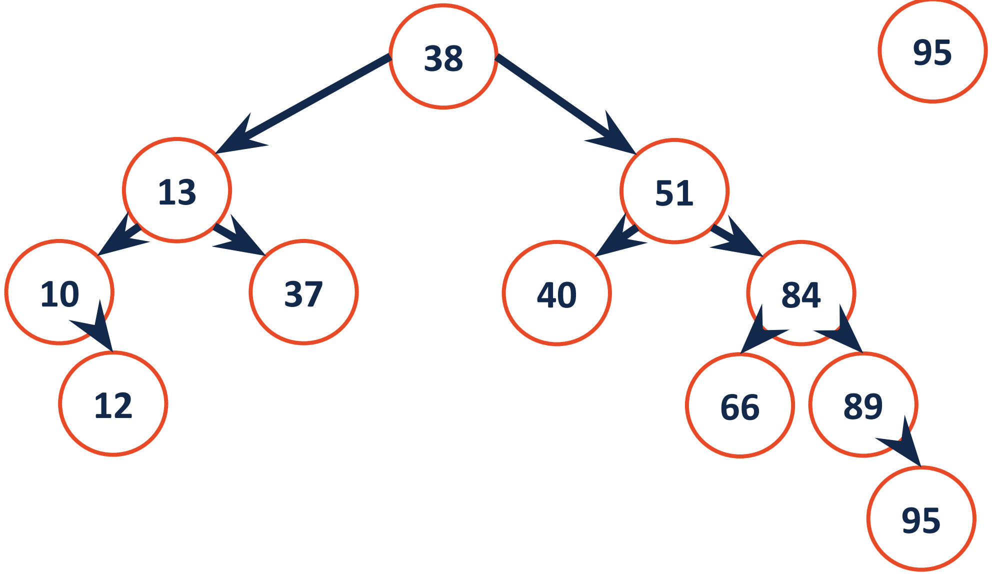
Removing element from BST




```
remove(40);
```

Removing element from BST



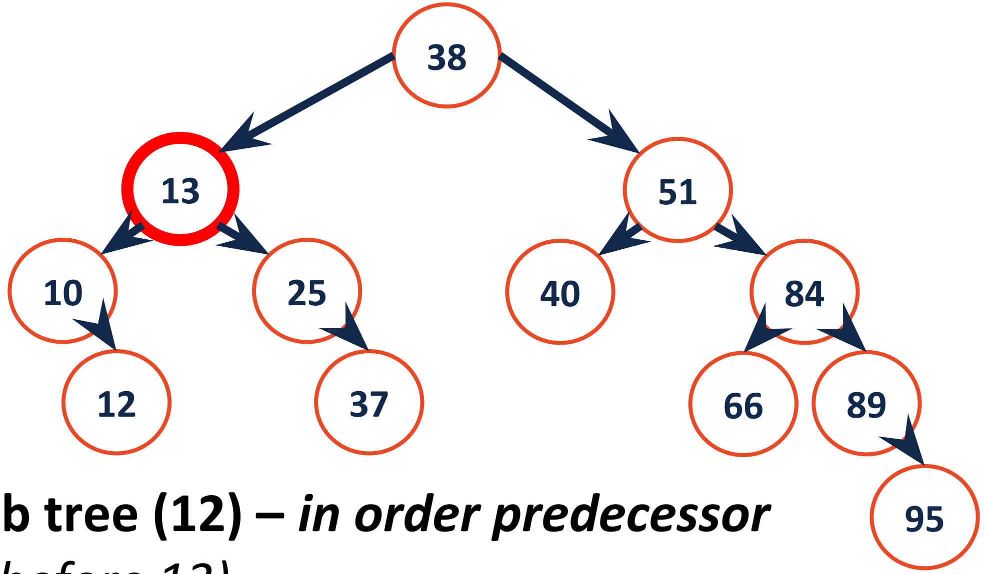



```
remove(25);
```

Removing element from BST:

Remove node with two children;

`remove(13);`



Find node to replace 13 so that BST properties still holds:

1. Maximum element in the left sub tree (12) – *in order predecessor*

(we do in order traversal 12 will be before 13)

2. Minimum element in the left sub tree (25) – in order successor

(when we do in order traversal 25 will be after 13)

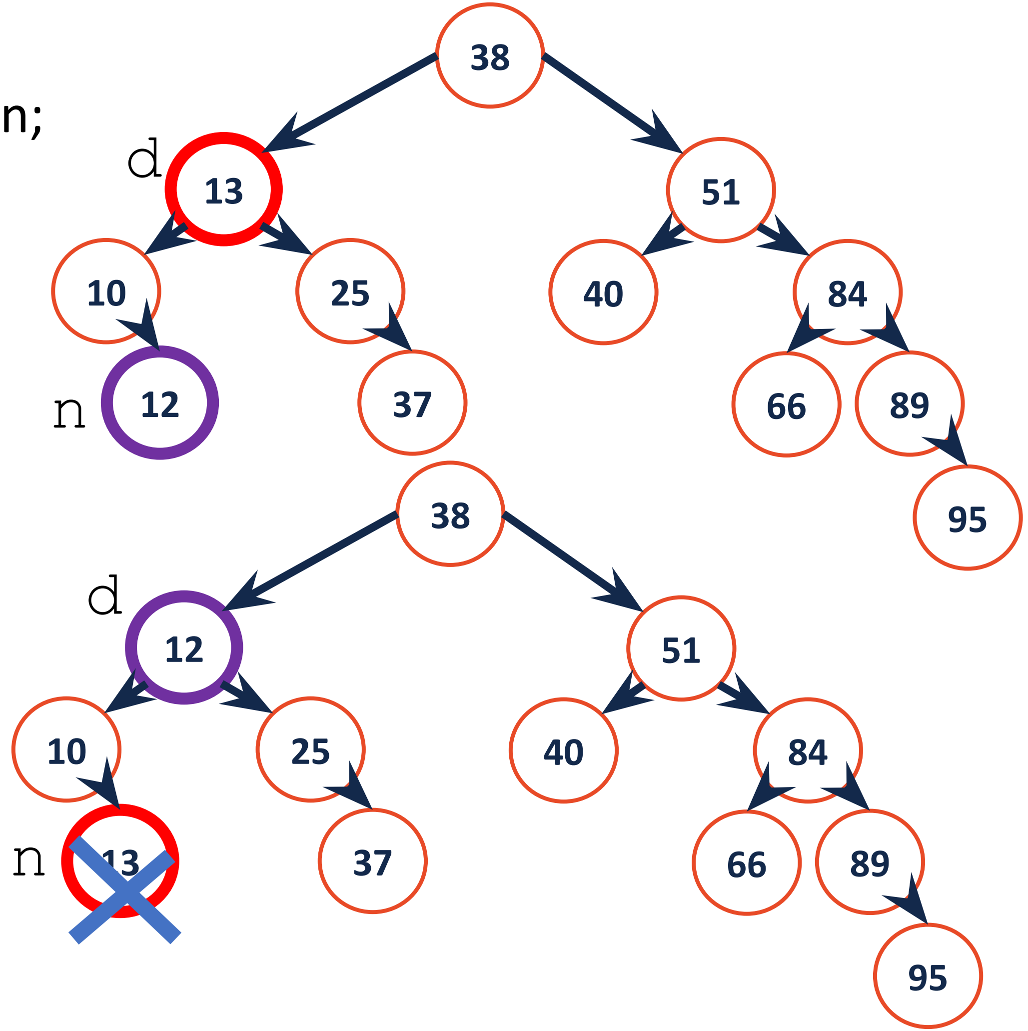
10 12 13 25 37 38 40 51 66 84 89 95

Removing element from BST:

Remove node with two children;

1. `Find node we want to delete (d)`
2. `Find IOP of the node (n)`
3. `Swap d with n`
4. `remove(n);`

`remove(n)` becomes simpler case (one child, or leaf node remove)



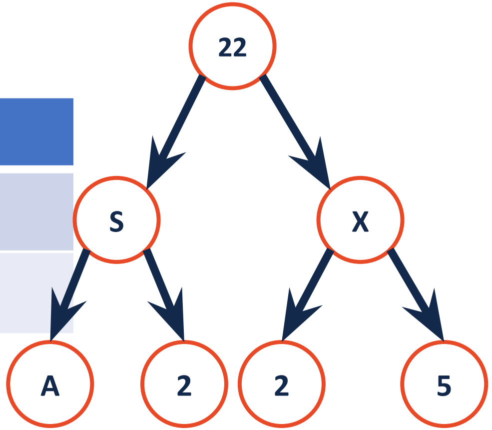
BST Analysis – Running Time

Operation	BST Worst Case
find	O(h)
insert	O(h)
delete	O(h)+O(h) = O(h)
traverse	O(n)

The relationship between h and n

We observe every operation on BST in terms of the height of the tree.
m(h) = max number of nodes of height h;

Height	m(h)
h =-1	0
h>-1	$1 + 2m(h-1) \;= 2^{h+1}O(2^h)$

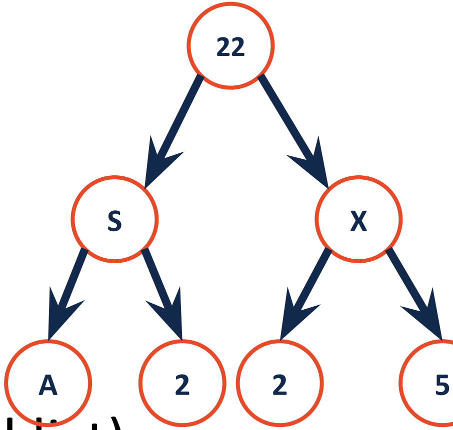


m(h-1) max number of nodes in the left/right subtree;

BST Analysis

Q: What is the minimum number of nodes in a tree of height h?

h+1 to create linked list like structure;



What is the maximum height for a tree of n nodes?

h <= n-1 (maximum height is reached with linked list)

BST Analysis

- Therefore, for all BST:

  Lower bound for height: $\boldsymbol{h = O(\lg n)}$

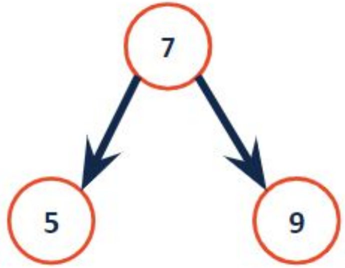
  Upper bound for height: $\boldsymbol{h = O(n)}$

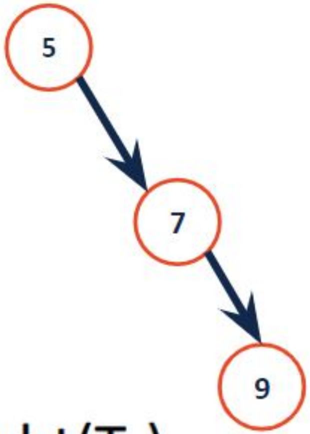
BST Analysis – Running Time

Operation	BST Average case	BST Worst case	Sorted array	Sorted List
find	$O(h) = O(\log n)$	$O(h) \leq O(n)$	Binary search $O(\log n)$	$O(n)$
insert	$O(h) = O(\log n)$	$O(h) \leq O(n)$	Find+shift data $O(n)$	$O(n)$
delete	$O(h) = O(\log n)$	$O(h) \leq O(n)$	$O(n)$	$O(n)$
traverse	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Height-Balanced Tree

- What tree makes you happier?





Height balance: $b = \text{height}(T_R) - \text{height}(T_L)$

A tree is height balanced if: $|b| \leq 1$

Useful Materials

- Lectures
- Lecture Notes
- Lecture/Lab worksheets
- Labs
- Q&A: https://docs.google.com/document/d/1zsHzVQVzwFjMtGQMroZDYG7WAKWQve0PZOWM6FaA0/edit#