# function templates (in reasonably logical syntax)

Definition of template

```
NAME ( parameter T ) is template {
    int f ( T x ) {
        return (int) x );
    }
}
```

Declaration

```
NAME ( float );
```

⇒ Definition

```
int f ( float x ) {
    return ( (int) x );
}
```

```
NAME ( double );
```

⇒

```
int f ( double x ) {
    return ( (int) x );
}
```

```
main ( ) {
    int n;
    float y;
    double z;

    n = f ( y );
    n = f ( z );
```

} function overloading

# Real C++ template syntax

Definition of template

```
template < typename T >
int f ( T x ) {
    return( (int) x );
}
```

Definition

```
int f ( float x ) {
    return ( (int) x );
}

int f ( double x ) {
    return ( (int) x );
}
```

no declaration

```
main ( ) {
    int n;
    float y;
    double z;

    n = f(y);
    n = f(z);
}
```

definition "specialization"

function overloading

## template overloading (for functions)

```
template < typename T >                          (1)
int f ( T x ) {  ...  }

template < typename T, typename W >              (2)
int f ( T x, W y ) {  ...  }

int f ( char * s ) {  ...  }                     (3)

int f ( int w ) {  ...  }                        (4)
```

```
main ( )

    int n
    float z
    double q

    f(z)      → (1)
    f(q)      → (1)
    f(z,q)    → (2)
    f("Hi")   → (3)
    f(n)      → (1) or (4) ?
```

```
template <typname T>
class C {
        C();
        int count();
        T A[100];
        int n;
};
```

template of class defn

defn of class w/ name "C<int>"

```
class C<int> {
        C<int>();
        int count();
        int A[100];
        int n;
};
```

```
template <typname T>
C<T>::C() { ... }
```

template of function defn

```
C<int>::C<int>() {
  ...
}
```

```
int C<T>::count() {
        return(n);
}
```

```
int C<int>::count() {
  ...
}
```

```
main()
  C<int> a;
  C<int> *pC;
  pC = &a;
```

Other parameters

```
template <typname T, int N>
class C2 {
        C();
        int count();
        T A[N];
        int n;
};
```

```
main()
  C2<int, 100> a;
```

Cannot overload class template as we could function templates

C2<int, 100> is not same class as C2<int, 10> nor C<int>

# Inheritance

## Override via specialization of Template

```
template <>
class C <char *>
{ ... };
```

no template declarations

```
C<char *>::C() {...}
C<char *>::f() {...}
```

## Override via partial specialization

```
template <typename T>
class C<T *> { ... }
```

```
template <typename T>
C<T *>::C() {...}
```

```
class D : public C<int> {...}
D::D() {...}
```

} class derived from template class

```
template <typename X>
class W : public C<X> {...}

template<typename X>
W<X>::W() {...}
```

} template class derived for template class

```
W<int> w;
```

---

**friends**

```
int f() {...}
```

f is a friend of R<int>, R<double>, ...

```
template <typename X>
class R {
    friend int f();
    ...
};
```

⇒ `friend int f( R<int> & )`

only friend of R<int>

implicit arg

also A::f()

# Exceptions

```
include <stdexcept>

class MyExcept : public runtime_error {
  public:
    MyExcept()
    ...
};

MyExcept::MyExcept()
  : runtime_error("Hi")
{...}

main()

try {
    doIt();
}
catch ( MyExcept & exception ) {
    cout << exception.what();
    fixOrDie( exception );
}

( catch ( runtime_exception & ) {
    exit(1);
} ) ?
```

other exceptions?

all derived classes

```
void doIt() {
  if ( cannotDoIt ) {
    throw MyExcept();
```

constructor call

```
  } else {
    doThing();
    ...
  }
```

might throw an exception;

unwind stack, look for "catch"

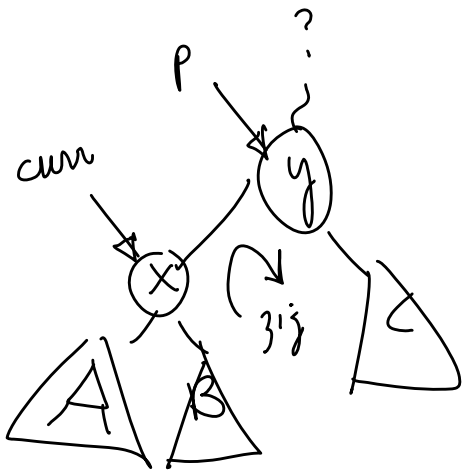
top: unexpected = terminate()

set_unexpected = f()

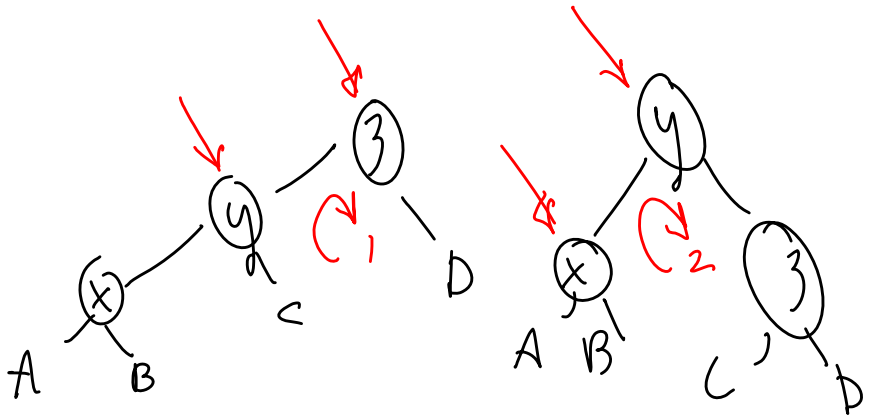
your own

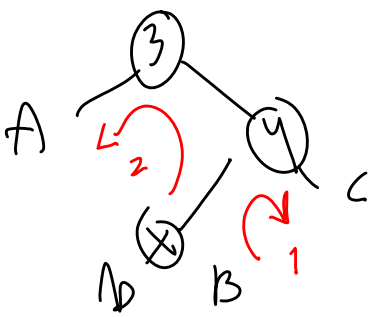
A[10]

A[0] = 7 = =




zig(curr, p)

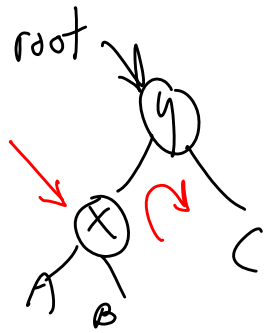



```
find(X) {
curr = Search(root)
 splay(curr)
}
```

splay ( curr )

if (curr == root) return;




if ( leftRoot )

zig ( curr )

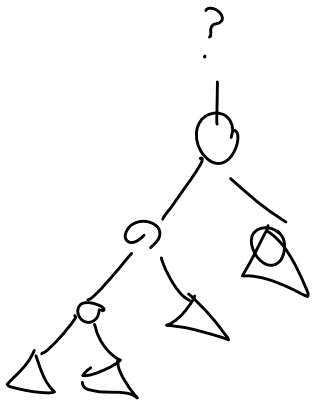
splay (curr)

if ( leftleft (curr))

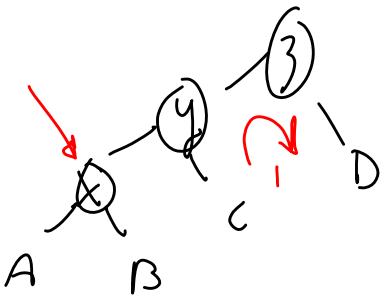
zig ( )

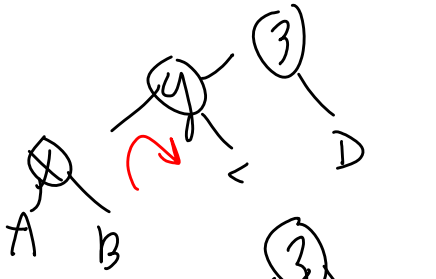
zig ( )

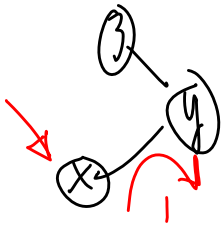
splay (curr)

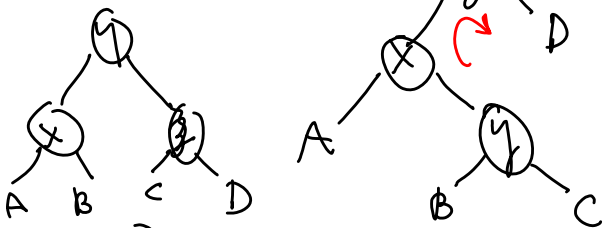



root = x



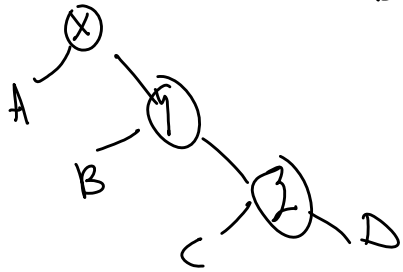



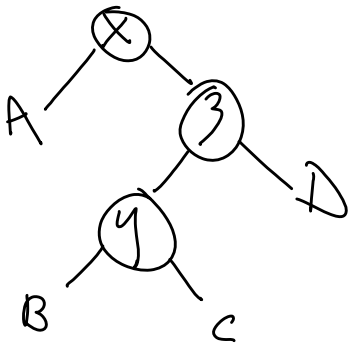









Thu

J 12:30

Clay 2

ch 3:30

Fri

1:30 Seth

3:00 Taiwo

Mon

10:00 Keelan