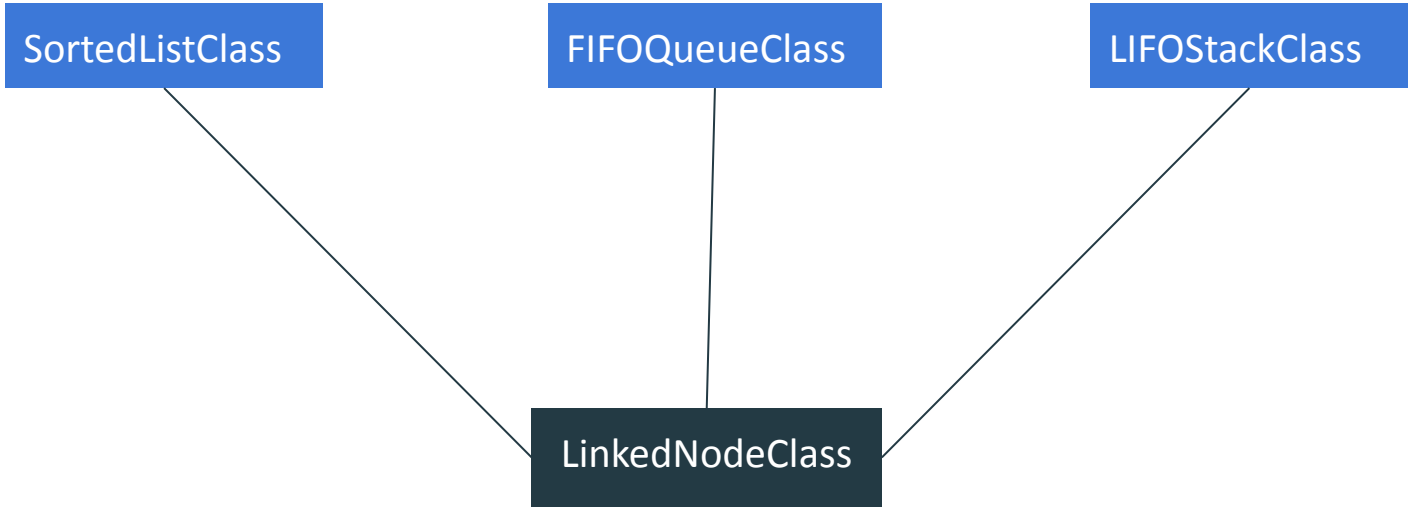
EECS402 Discussion 11!

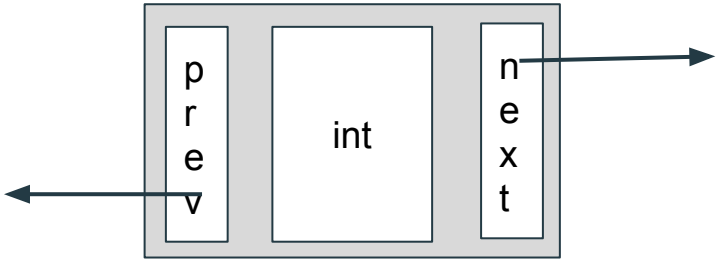
P4, Templates, Recursion, Friends, Exceptions

Project 4 Overview

Elements


SortedListClass
FIFOQueueClass
LIFOStackClass
LinkedNodeClass

LinkedNodeClass



Attributes: value, prev pointer, next pointer

Methods:

- Constructor- (prev, value, next)
- getValue, getNext, getPrev
- setNextPointerToNull, setPreviousPointerToNull
- setBeforeAndAfterPointers (read spec carefully for this one)

SortedListClass

Attributes: head, tail

Methods:

Look at the last Discussion(10)

Default ctor, copy ctor, dtor, clear

InsertValue, removeFront, removeLast

printForward, printBackward

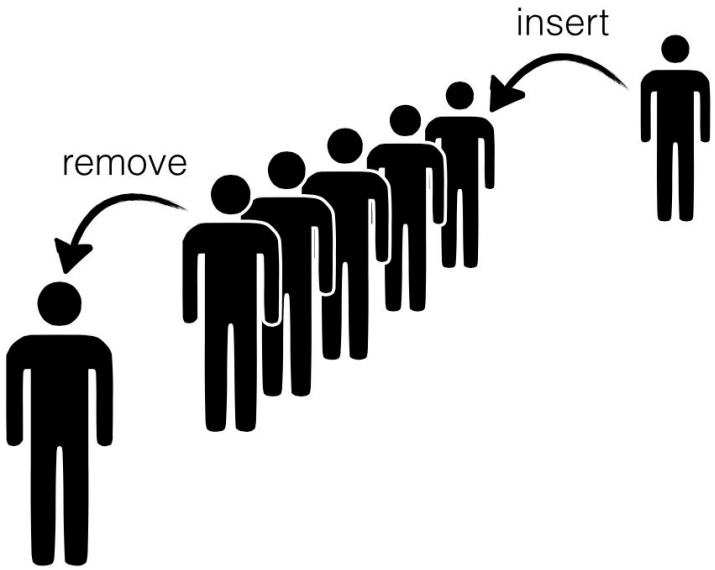
getNumElems, getElemAtIndex

FIFOQueueClass

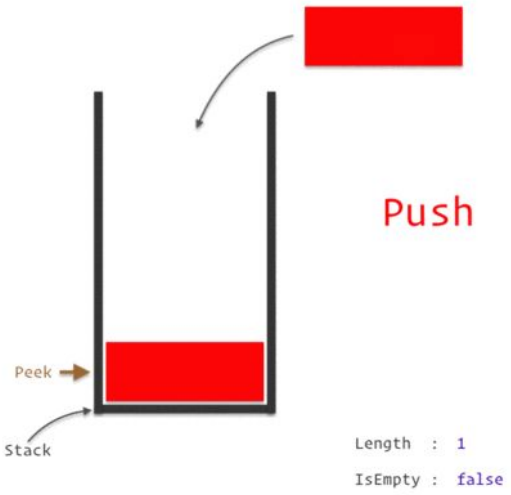
Attributes: head, tail

Methods:

- Default ctor, dtor, clear
- enqueue, dequeue
- print,
- getNumElems



LIFOStackClass



Attributes: head, tail

Methods:

- Default ctor, dtor, clear
- Push, pop Always push and pop on the same end
- Print
- getNumElems

General Tips

- Follow the spec!! Make sure everything is named *exactly* right
- Draw things out
- Start early!!!
- Check all edge cases

Templates

Templates

We would use this for Project 5

- Useful when we don’t know the type of a variable we want to use

- Allows for code to be more versatile

Templates

```
6   template <class T>
7   T maxElt(T val1, T val2) {
8       if(val1 > val2){
9           return val1;
10      } else {
11          return val2;
12      }
13  }
14
15
16  int main(){
17
18      cout << maxElt(2, 3) << endl;
19      cout << maxElt( "hi", "hello") << endl;
20      cout << maxElt(3.4, 1.2) << endl;
21  }
```

Templates

```
6   template <class T>
7   T maxElt(T val1, T val2) {
8       if(val1 > val2){
9           return val1;
10      } else {
11          return val2;
12      }
13  }
14
15
16  int main(){
17
18      cout << maxElt(2, 3) << endl;
19      cout << maxElt( "hi", "hello") << endl;
20      cout << maxElt(3.4, 1.2) << endl;
21  }
```

3
hi
3.4

Template Example

Write a function that adds two variables of an unknown type and returns the sum

Assume the variable type supports the + operation

Solution

```
template <class T>
T add(T val1, T val2) {
    return val1 + val2;
}
```

Template Example

Write a function that swaps two variables of an unknown type

Template Solution

```
template <class T>
void swapElts(T & val1, T & val2) {
    T temp = val1;
    val1 = val2;
    val2 = temp;
}
```

Template Classes

- Very similar to templated functions
- Allows classes to be more versatile
- Need to specify what type the template is when creating an instance of the class

```
6   template <class T>
7   class MyClass {
8       private:
9           T value;
10      public:
11          MyClass(T val) {
12              value = val;
13          }
14
15          T getValue() {
16              return value;
17          }
18  };
19
20
21  int main(){
22
23      MyClass<int> tempClass(5);
24
25      cout << tempClass.getValue() << endl;
26
27  }
```

Template Class Example

Create a templated class with

Attributes: 2 ‘mystery items’ named item1 and item2 (they can be different types)

Member functions: getFirstItem, getSecondItem, printItems, value constructor

Example Solution

```cpp
template <class T, class T2>
class MysteryItems {
    private:
        T item1;
        T2 item2;
    public:

        MysteryItems(T val1, T2 val2) {
            item1 = val1;
            item2 = val2;
        }

        T getFirstItem() {
            return item1;
        }

        T2 getSecondItem() {
            return item2;
        }

        void printItems() {
            cout << "Item 1: " << item1 << endl;
            cout << "Item 2: " << item2 << endl;
        }
};
```

This Pointer

This

- ‘this’ is used as a pointer to the object that a function was called on

- It’s passed as an implicit parameter to every member function

```
int getOunces(){
    return ounces;
}
```

==

```
int getOunces(){
    return this->ounces;
}
```

This

- ‘this’ is used as a pointer to the object that a function was called on

- It’s passed as an implicit parameter to every member function

```
Test& Test::func ()
{
    // Some processing
    return *this;
}
```

Friend functions

“Friendship is given, not taken.”

Example in Pair class

- Allows us to avoid the print() function which feels less natural than the extraction operator.
- Necessary because the left side of the operator is an ostream object, not our object
- Note: We really don't use friend functions / classes hardly ever

```
template < class T1, class T2 >
class Pair {
private:
    T1 first;
    T2 second;

public:
    Pair(T1 firstInput, T2 secondInput)
    : first(firstInput), second(secondInput) {}

    void print();

    void setFirst(T1 newFirst);

    void setSecond(T2 newSecond);

    void set(T1 newFirst, T2 newSecond);

    friend ostream& operator<<(ostream& os, const Pair<T1, T2>& rhs);
};
```

For project 5 as well

```
template < class T1, class T2 >
ostream& operator<<(ostream& os, const Pair<T1, T2>& rhs) {
    os << "( " << rhs.first << ", " << rhs.second << " )" << endl;
    return os;
}
```

Recursion

If u hate it just don’t use this

Two parts of a recursive function

```
int factorial(int n)
{
    if(n > 1)
        return n * factorial(n - 1);
    else
        return 1;
}
```

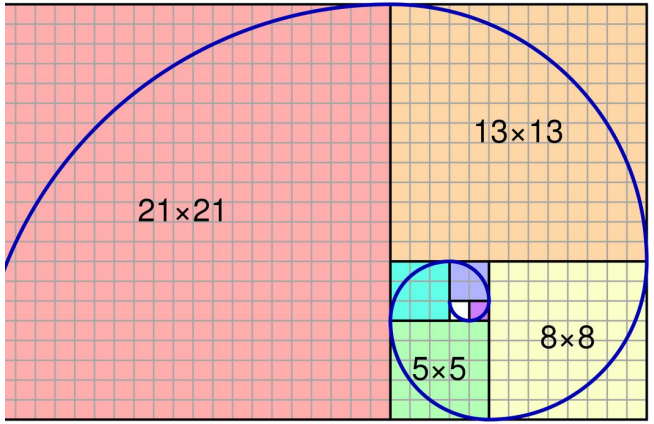
If no return 1, this will never stop

- Base case(s): The set of inputs where there is a single specific answer that you return directly to the user.
- Recursive equation: The way the new smaller problem relates to old problems

Recursive functions allow you to break the problem into smaller subproblems

Example: Fibonacci

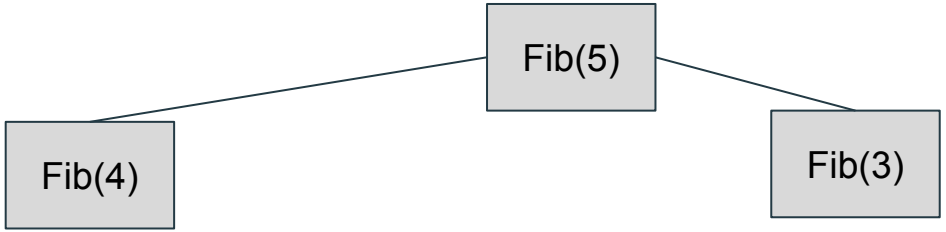
Write a function that returns the nth number in the fibonacci sequence

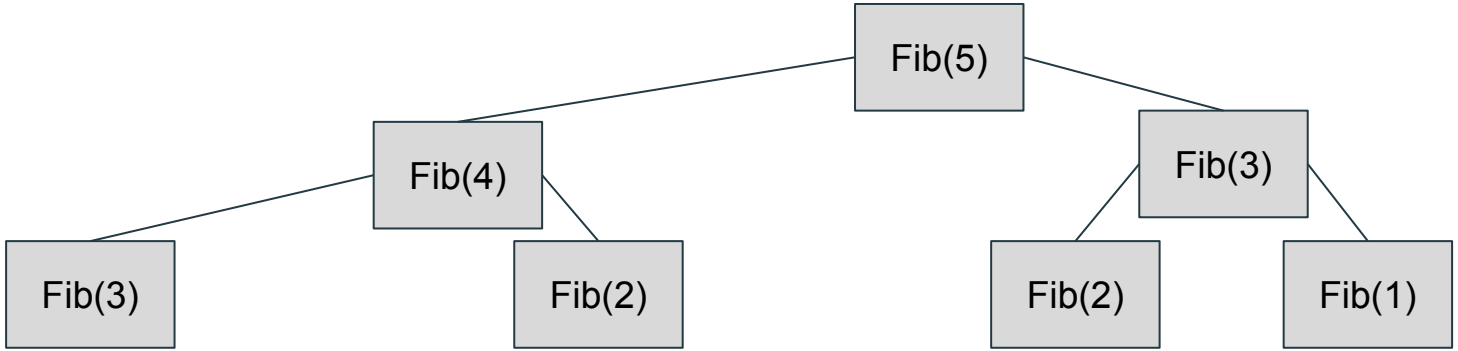


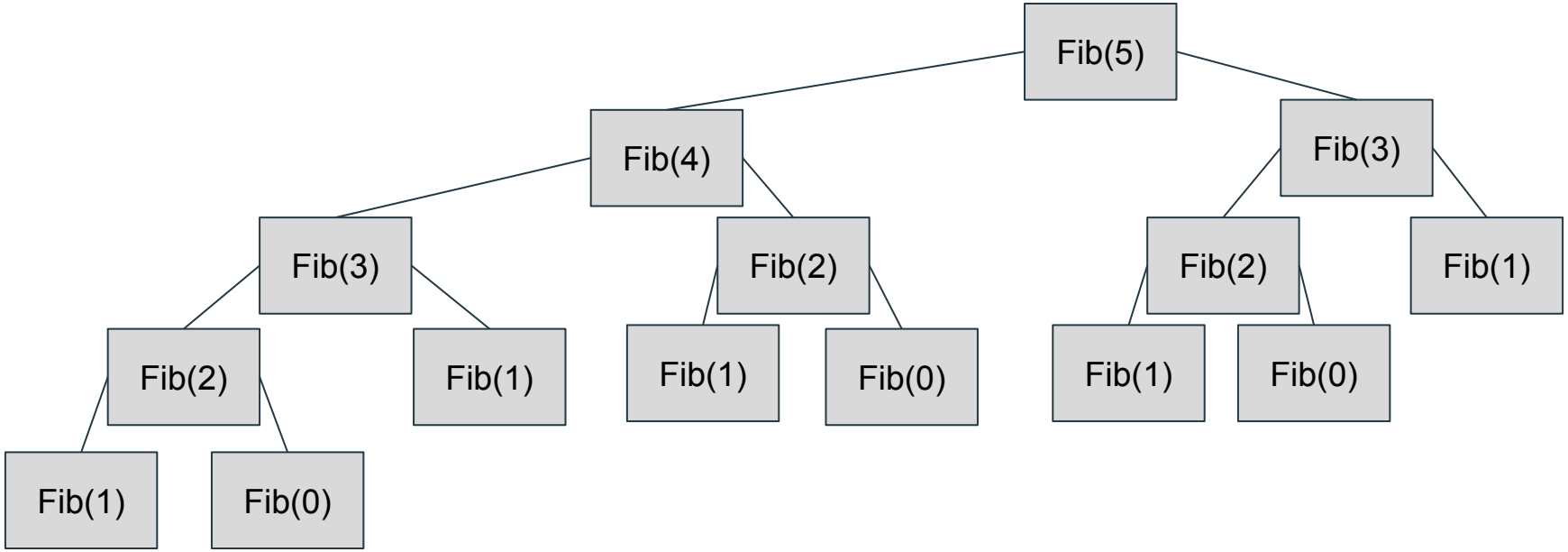
1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2	**13+21=34**
1+2=3	**21+34=55**
2+3=5	**34+55=89**
3+5=8	**55+89=144**
5+8=13	**89+144=233**
8+13=21	**144+233=377**

Fib(5)


Fib(5)
Fib(4)
Fib(3)


Fib(5)
Fib(4)
Fib(3)
Fib(3)
Fib(2)
Fib(2)
Fib(1)


Fib(5)
Fib(4)
Fib(3)
Fib(3)
Fib(2)
Fib(2)
Fib(1)
Fib(2)
Fib(1)
Fib(1)
Fib(0)
Fib(1)
Fib(0)
Fib(1)
Fib(0)

Example Solution

```
int fib(int n) {
    if (n <= 1){
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

Exceptions

Exceptions

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

- Use "catch(...)" to catch any other unspecified types!

```
const int TOO_LOW_TYPE = 1;
const int TOO_HIGH_TYPE = 2;
class IndexExcepClass
{
  public:
    IndexExcepClass(int inType)
    {
       excepType = inType;
    }
    void print()
    {
      if (excepType == TOO_LOW_TYPE)
      {
        cout << "Index too low!";
      }
      else if (excepType == TOO_HIGH_TYPE)
      {
        cout << "Index too high!";
      }
    }
   private:
    IndexExcepClass()
    { ; }

    int excepType;
};
```

```
void decrAryElem(int ary[], int ind)
{
  if (ind < MINARY)
    throw IndexExcepClass(TOO_LOW_TYPE);
  if (ind > MAXARY)
    throw IndexExcepClass(TOO_HIGH_TYPE);

  ary[ind]--;
}
```

From main:

```
  for (i = -1; i <= MAXARY + 1; i++)
  {
    try
    {
      decrAryElem(myAry, i);
    }
    catch (IndexExcepClass iex)
    {
      iex.print();
      cout << endl;
    }
    cout << "Continuing!" << endl;
  }
```

```
Index too low!
Continuing!
Continuing!
Continuing!
Continuing!
Index too high!
Continuing!
```