

CS2040C

Data Structures and Algorithms

Welcome!

Today's Plan

- Abstract Data Type
 - Stack
 - List
 - Queue
- Inheritance

How do you decide what to cook?

- See what ingredient you have in your fridge
- See what can you cook
- Look up some recipes
- Modify/improvise during cooking

Result



How does a 5-stars hotel decides

- The Christmas Dinner Menu?



How does a 5-stars hotel decides

- The Christmas Dinner Menu?
 - They decide what will be on the menu first
 - Then they find the
 - recipe
 - ingredient
 - cook
- And usually the manager who design the menu
 - Doesn't care too much about how to buy the ingredient, where to find the cook, how is the food cooked

Difference

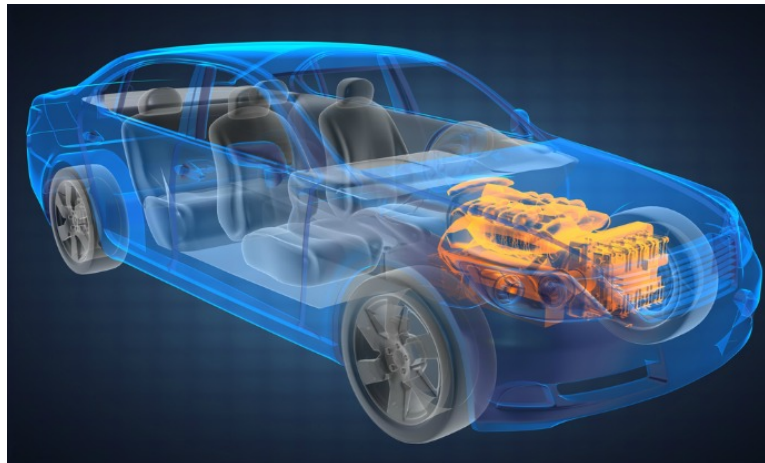
- How hotel decides what's on the menu
 - Design what is the outcome first
 - How we cook
 - See what can you cook
 - Look up some recipes
 - what ingredient in the fridge
 - Modify/improvise during cooking
- What customers will get
- Implementation

Cars

- Interface



- Implementation



Abstract Data Types

- Interface/behavior
 - How the others use/communicate with it
- Implementation
 - The details of how you implement it

Abstract Data Types: Symbol Table

- Interface/behavior

<code>void</code>	<code>insert(Key k, Value v)</code>	<i>insert (k,v) into table</i>
<code>Value</code>	<code>search(Key k)</code>	<i>get value paired with k</i>
<code>void</code>	<code>delete(Key k)</code>	<i>remove key k (and value)</i>
<code>boolean</code>	<code>contains(Key k)</code>	<i>is there a value for k?</i>
<code>int</code>	<code>size()</code>	<i>number of (k,v) pairs</i>

- Implementation

- Use Linked List? Hash Tables? Trees?

Abstract Data Types

Specification:

- Interface
- Behavior

Implementation:

- Algorithm
- State

The Turk

- Our story starts in the year 1770 in the Kingdom of Hungary within the Habsburg Empire. There, Wolfgang von Kempelen is building a machine capable of playing the game of chess. His plan is to compete against the best chess players of the time.
- With his machine finished, he impresses the court of Maria Theresa of Austria. Kempelen and his chess playing automaton are quickly becoming famous, defeating most of their opponents during demonstrations around Europe. The audience includes statesmen such as Napoleon Bonaparte and Benjamin Franklin.



The Turk

- The machine, seemingly operated by a torso and head dressed in Turkish robes was, in fact, a mechanical illusion. It was controlled by a human operator hidden inside. The Turk was a very elaborate parlor trick. A hoax, designed to make others think they are competing against a real machine. The secret was fully revealed only in 1850



Plate 4.

The Turk

- Ever since we use the term “*Mechanical Turk*” for a system that appears to be autonomous but in fact, it needs a human aid to operate

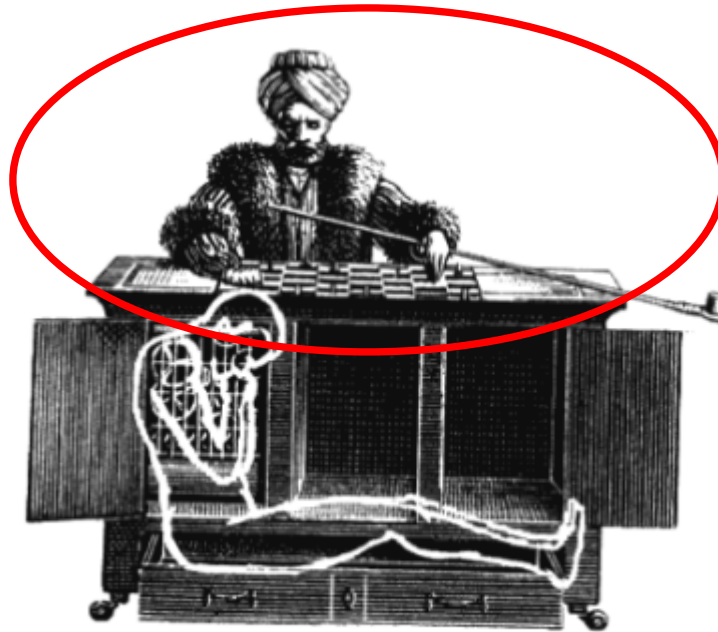


Plate 3.

- So this “Turk machine” is only an **interface**!



Abstract Data Types

Stack

Interface:

- void push(element x)
- element pop()

Behavior: (LIFO: last-in, first-out)

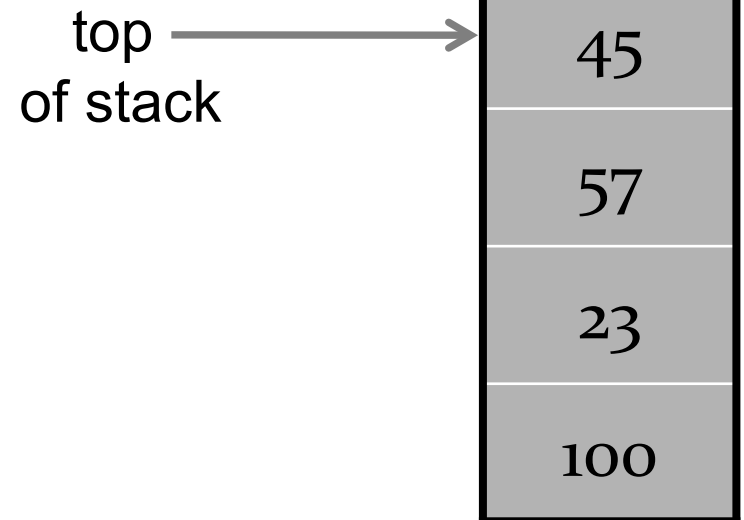
- push(x) : adds element x to the stack
- pop() : removes the mostly recently added element and returns it

Abstract Data Types

Stack

Interface:

- void push(element x)
- element pop()
- empty()

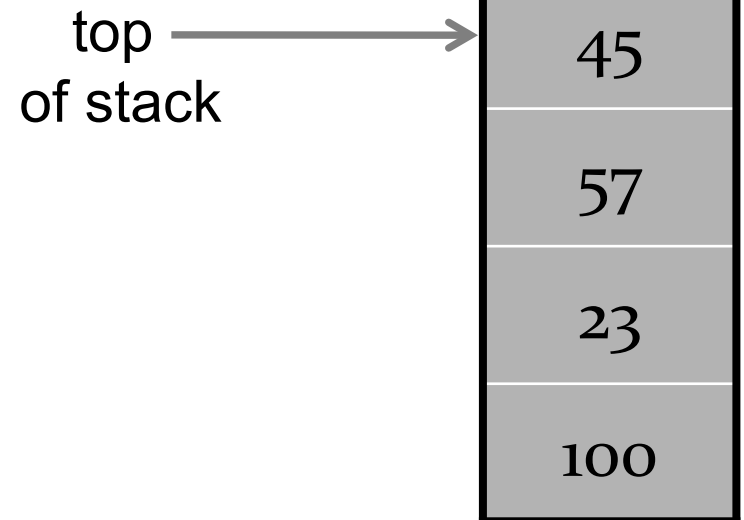


Abstract Data Types

Stack

Execution:

- `push(77)`

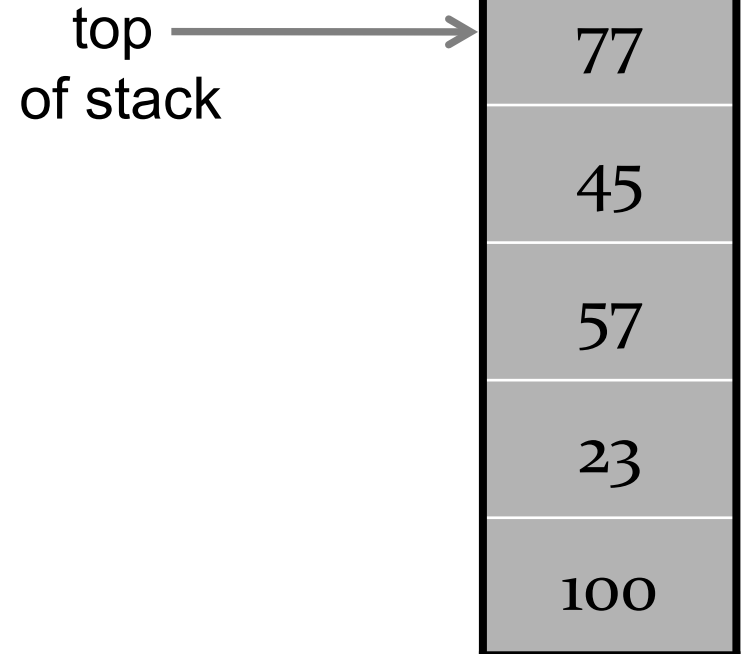


Abstract Data Types

Stack

Execution:

- `push(77)`



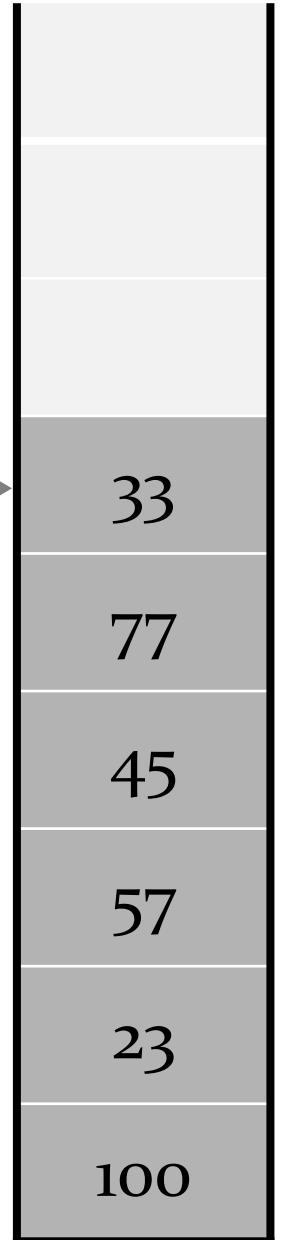
Abstract Data Types

Stack

Execution:

- push(77)
- push(33)

top
of stack



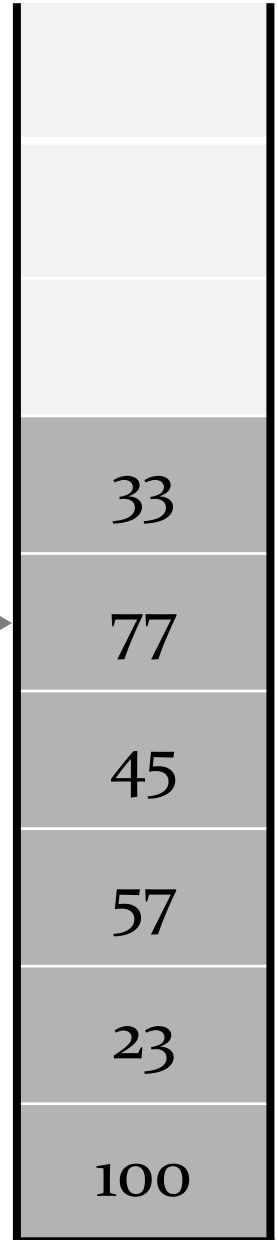
Abstract Data Types

Stack

Execution:

- push(77)
- push(33)
- pop() → ??

top
of stack →



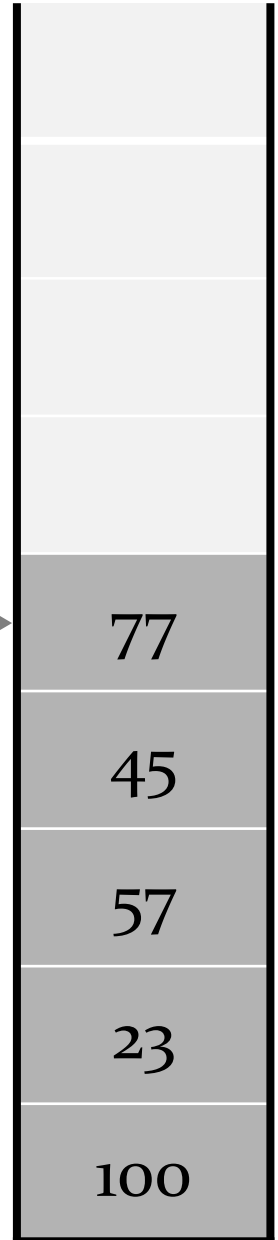
Abstract Data Types

Stack

Execution:

- push(77)
- push(33)
- pop() → 33

top
of stack →



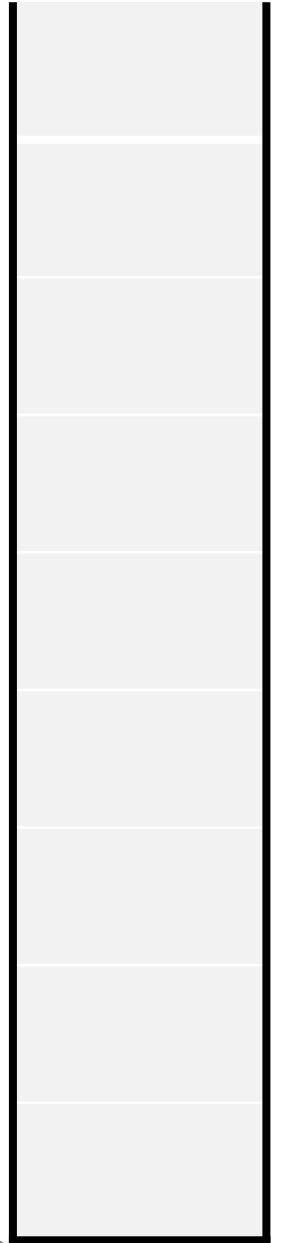
Abstract Data Types

Stack

Execution:

- $\text{pop()} \rightarrow 23$
- $\text{pop()} \rightarrow 100$

top
of stack →



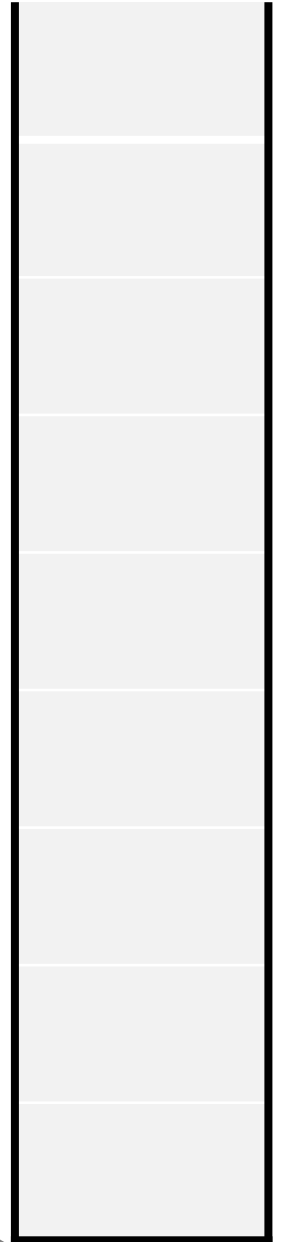
Abstract Data Types

Stack

Execution:

- $\text{pop()} \rightarrow 23$
- $\text{pop()} \rightarrow 100$
- $\text{pop()} \rightarrow ??$

top
of stack 



Abstract Data Types

Stack

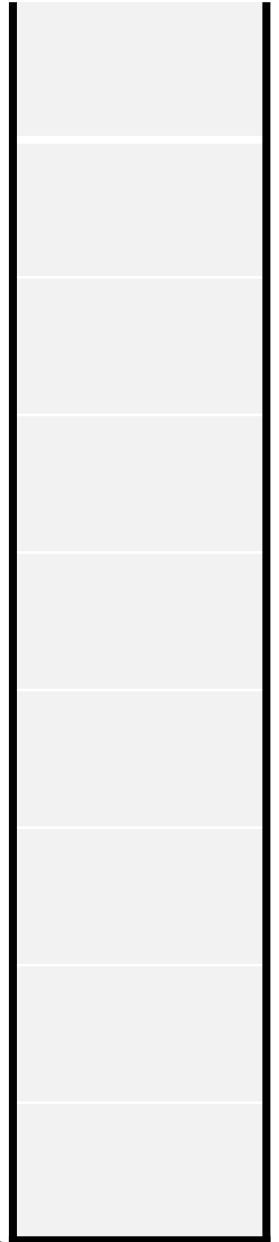
Execution:

- `pop()` \rightarrow 23
- `pop()` \rightarrow 100
- `pop()` \rightarrow ??

- **Error!**

- Option 1: throw exception (*postponed*)
- Option 2: modify specification

top
of stack 



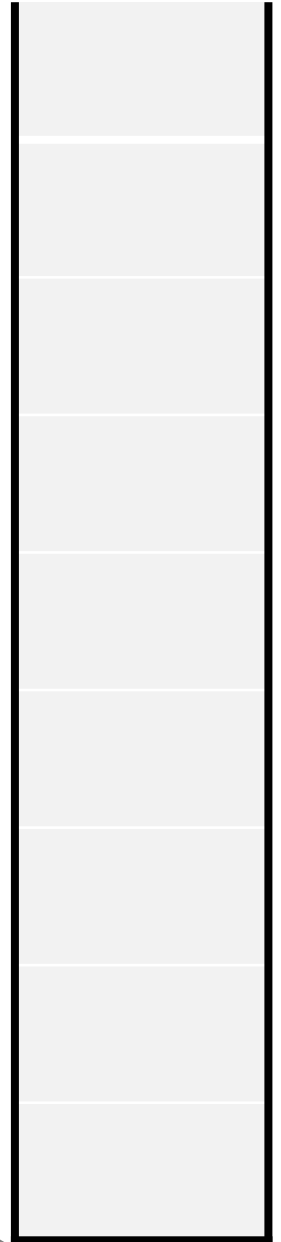
Abstract Data Types

Stack


Execution:

- `pop()` \rightarrow 23
- `pop()` \rightarrow 100
- `empty()` \rightarrow true

top
of stack 



Implementation of an ADT

-  How do you implement a Stack?
- Hint:
 - The skeleton code for Assignments 1 and 2 are almost done for a stack!

Implementation of the ADT Stack

- Any idea?

```
template <class T>
class Stack {
private:
    List<T> _ll;
```

```
public:
    void push(T x);
    T pop();
    bool isEmpty();
};
```

Implementation of the ADT Stack

- Any idea?

```
template <class T>
void Stack<T>::push(T x) {
    _ll.insertHead(x);
}
```

```
template <class T>
T Stack<T>::pop() {
    T item = _ll.headItem();
    _ll.removeHead();
    return item;
}
```

Abstract Data Types

Queue

Interface:

- void enqueue(element x)
- element dequeue()

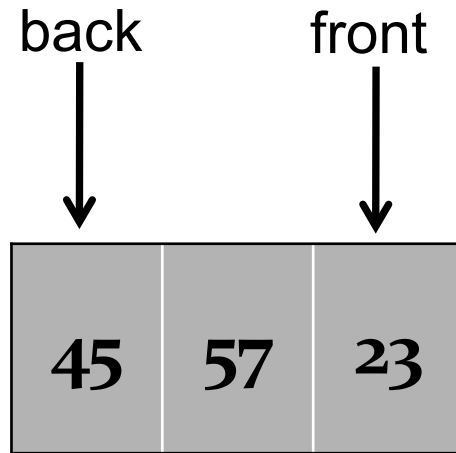
Behavior: (FIFO: first-in, first-out)

- enqueue(x) : adds element x to the front of the queue
- dequeue() : removes and returns element at the end of the queue

Abstract Data Types

Queue

Execution:

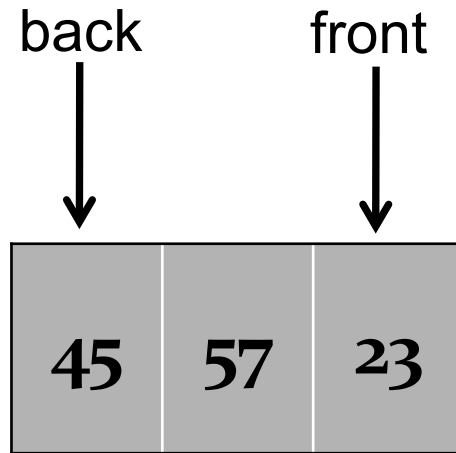


Abstract Data Types

Queue

Execution:

- `enqueue(7)`

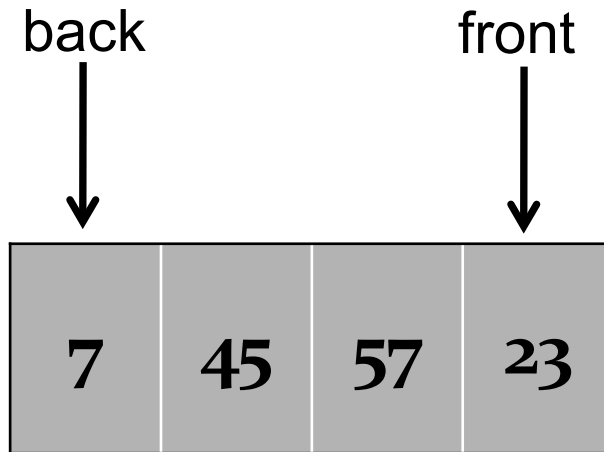


Abstract Data Types

Queue

Execution:

- enqueue(7)

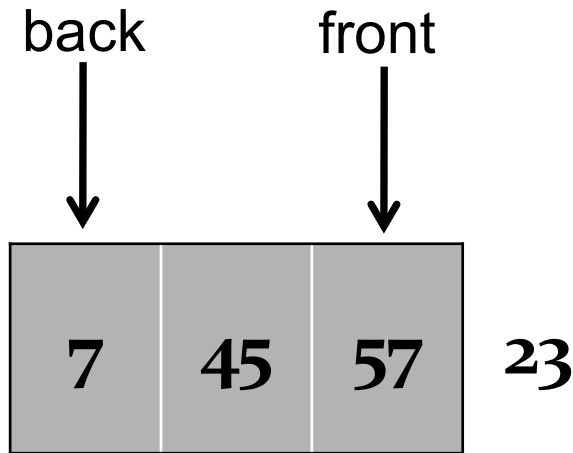


Abstract Data Types

Queue

Execution:

- enqueue(7)
- dequeue() → 23



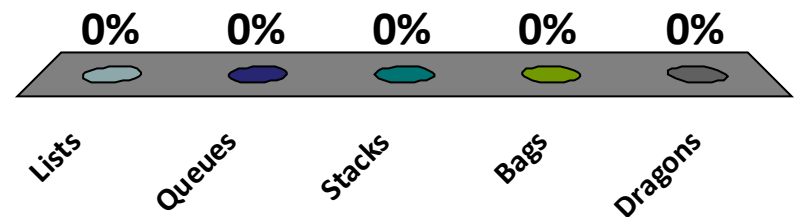
Implementation of ADT Queue?

-  Let you think about it

Which abstract data type appears most frequently in practice?

- a. Lists
- b. Queues
- ✓ c. Stacks
- d. Bags
- e. Dragons

Response
Counter

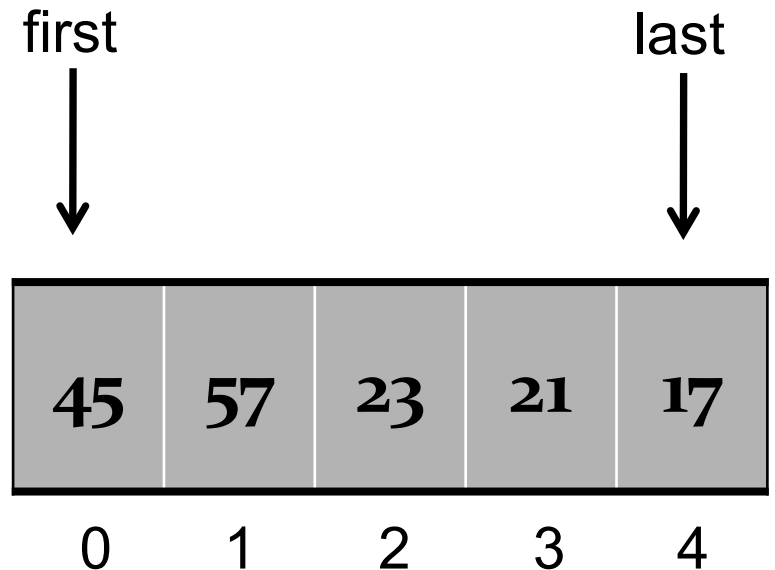


Abstract Data Types

List

Interface:

```
void append(int x)
void prepend(int x)
void put(int x, int slot)
void remove(int x)
int getFirst()
int getLast()
int get(int slot)
boolean isEmpty()
```



Inheritance



INHERITANCE

Christopher Paolini

Building a better stack

What if I want to build a better stack?

- Add functionality
- Improve efficiency
- Or I just want the computer to play a “bee” sound when something is pushed and a “do” sound when something is popped.....
- But someone thinks that’s annoying and wants to use the original stack..




Building a better stack

What if I want to build a better stack?

- Add functionality
- Improve efficiency

Solutions:

- Implement from scratch
 - Modify original class
 -  Copy-paste old code to new class
- waste time fixing old bugs
- Editing the class directly means other people who do not want the implementation will suffer

Building a better stack

What if I want to build a better stack?

- Option 1: implement stack **again**

```
template <class T>
class BeeBooStack {
    // implement everything again
}
```

- Useful when:

Entirely new implementation (e.g., don't use an array, use fractional cascading on a buffered tree).

Building a better stack

Inheritance

- BeeBooStack is a **subclass** (child) of Stack
- Stack is the **superclass**(parent) of BeeBooStack

```
template <class T>
class BeeBooStack public Stack<T> {

public:
    void push(T);
    T pop();

};
```

Building a better stack

Inheritance

- Subclass has all the functionality of the parent!

```
BeeBooStack<Food> s_food;  
  
if (s_food.isEmpty())  
    cout << "I am hungry" << endl;
```

Building a better stack

Inheritance

- Subclass can override parent class

```
template <class T>
class BeeBooStack:public Stack<T> {

public:
    void push(T);
    T pop();

};
```

Building a better stack

Inheritance

- Subclass can override parent class

```
template <class T>
void BeeBooStack<T>::push(T x) {
    cout << "Bee" << endl;
    Stack<T>::push(x);
}

template <class T>
T BeeBooStack<T>::pop() {
    cout << "Boo" << endl;
    return Stack<T>::pop();
}
```



Building a better stack



Inheritance

- Subclass can override parent class

```
BeeBooStack<Food> s_food;  
Food food1("Salad", 100);  
Food food2("Chicken", 200);  
Food food3("Curry", 40);  
Food food4("Ice Cream", 300);
```

```
s_food.push(food1);  
s_food.push(food2);  
s_food.push(food3);  
s_food.push(food4);  
cout << s_food.pop() << endl;  
s_food.push(food1);  
cout << s_food.pop() << endl;  
cout << s_food.pop() << endl;  
cout << s_food.pop() << endl;
```

```
Bee  
Bee  
Bee  
Bee  
Boo  
Ice Cream with 300 calories  
Bee  
Boo  
Salad with 100 calories  
Boo  
Curry with 40 calories  
Boo  
Chicken with 200 calories
```

Polymorphism

- What if

```
Stack<int> *s;  
  
BeeBooStack<int> bbsi;  
  
s = &bbsi;  
  
s->push(10);
```

- Which “push” will be called?
 - Stack<int> , or
 - BeeBooStack<int> ?

Polymorphism

- What if

```
Stack<int> *s;  
  
BeeBooStack<int> bbsi;  
  
s = &bbsi;  
  
s->push(10);
```

- It will call
 - `Stack<int>::push()`
- But it doesn't make sense!?

Polymorphism

- A power technique when you make the function virtual

```
template <class T>
class Stack {

public:
    virtual void push(T x);
    virtual T pop();
    virtual bool isEmpty();

};
```

Polymorphism

```
template <class T>
class Stack {

public:
    virtual void push(T x);
    virtual T pop();
    virtual bool isEmpty();

};
```

```
Stack<int> *s;

BeeBooStack<int> bbsi;

s = &bbsi;

s->push(10);
```

- s->push(10) will call the push from BeeBooStack<int>

Polymorphism

```
class Animal {
public:
    virtual void talk()
    { cout << "*Nothing*" << endl;
    };

class Dog :public Animal {
public:
    virtual void talk()
    { cout << "Woof" << endl; }
};

class Cat :public Animal {
public:
    virtual void talk()
    { cout << "Meow" << endl; }
};
```

```
Animal *dolly = new Dog();
Animal *orange = new Cat();

dolly->talk();
orange->talk();
```

```
Woof
Meow
Press any key to continue . . .
```

Polymorphism

- Another exmaple:
- Parent class: Shape
 - `calArea()`
 - `drawOnScreen()`
 - `scale(x,y)`
- Children classes:
 - Circle, Triangle, Rectangles, etc.

Access Control in Inheritance

- What if

```
template <class T>
T BeeBooStack<T>::pop() {

    if (_ll.empty())
        cout << "No more item to pop!" << endl;

    cout << "Boo" << endl;
    return Stack<T>::pop();
}
```

- Compilation Error!

- error C2248: 'Stack<Food>::_ll': cannot access private member declared in class 'Stack<Food>'

Access Control in Inheritance

- In the parent class

```
template <class T>
class Stack {
private:
    List<T> _ll;

public:
    virtual void push(T x);
    virtual T pop();
    virtual bool isEmpty();
};
```

- the variable `_ll` is private

Access Control in Inheritance

- Let's face it
 - A child class cannot access the **private** member/method of its parent class
- But what if the child really need to access some members of the parents, that are NOT public?

```
template <class T>
class Stack {
protected:
    List<T> _ll;

public:
    virtual void push(T x);
    virtual T pop();
    virtual bool isEmpty();
};
```