COMP2521 24T1
Abstract Data Types

Kevin Luxa

abstraction
abstract data types
stacks and queues
sets
Abstraction

is the process of hiding or generalising the details of an object or system to focus on its high-level meaning or behaviour.
Assembly languages abstract away machine code

```
0000000000000000 <fn>:
push rbp
  0: 55  push rbp
  1: 48 89 e5  mov rbp, rsp
  4: 89 7d ec  mov DWORD PTR [rbp-0x14], edi
  7: c7 45 fc 01 00 00 00  mov DWORD PTR [rbp-0x04], 0x1
 e: c7 45 f8 01 00 00 00 00  mov DWORD PTR [rbp-0x08], 0x1
15: eb 0e  jmp 25 <fn+0x25>
17: 8b 45 fc  mov eax, DWORD PTR [rbp-0x04]
1a: 0f af 45 f8  imul eax, DWORD PTR [rbp-0x08]
1e: 89 45 fc  mov DWORD PTR [rbp-0x04], eax
21: 83 45 f8 01  add DWORD PTR [rbp-0x08], 0x1
25: 8b 45 f8  mov eax, DWORD PTR [rbp-0x08]
28: 3b 45 ec  cmp eax, DWORD PTR [rbp-0x14]
2b: 7e ea  jle 17 <fn+0x17>
2d: 8b 45 fc  mov eax, DWORD PTR [rbp-0x04]
30: 5d  pop rbp
31: c3  ret
```
Modern programming languages abstract away assembly code

```c
int fn(int n) {
    int res = 1;
    for (int i = 1; i <= n; i++) {
        res *= i;
    }
    return res;
}
```
A function abstracts away the details or steps of a computation.
We drive a car by using a steering wheel and pedals

We operate a television through a remote control

We deposit and withdraw money to/from our bank account via an ATM
To use a system, it should be enough to understand *what* its components do without knowing *how*...
A data type is...

- a collection or grouping of values
  - could be atomic, e.g., int, double
  - could be composite/structured, e.g., arrays, structs
- a collection of operations on those values

Examples:
- int
  - operations: addition, multiplication, comparison
- array of ints
  - operations: index lookup, index assignment
An abstract data type...

is a description of a data type from the point of view of a user, in terms of the operations on the data type and the behaviour of these operations.

Importantly, an ADT does not specify how the data type or operations should be implemented.
Example of an ADT: Stack

A stack is a linear collection of items with two main operations:

- **push**: adds an item to the top of the stack
- **pop**: removes the item at the top of the stack
Abstract Data Types

Example

User

Stack

Operations

**push**
adds an item to the top of the stack

**pop**
removes the item at the top of the stack
Abstract Data Types

Example

User
push 8
push 3
push 7

Stack

Operations

push
adds an item to the top of the stack

pop
removes the item at the top of the stack
Abstract Data Types

Example

User
push 8
push 3
push 7
pop
pop
push 1

Stack

Operations

push
adds an item to the top of the stack

pop
removes the item at the top of the stack
Abstract Data Types

Example

User
- push 8
- push 3
- push 7
- pop
- pop
- push 1

Stack

Operations

**push**
adds an item to the top of the stack

**pop**
removes the item at the top of the stack
Abstract Data Types

Example

<table>
<thead>
<tr>
<th>User</th>
<th>Stack</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>push 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push 7</td>
<td>8</td>
<td><strong>push</strong> adds an item to the top of the stack</td>
</tr>
<tr>
<td>pop</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td>3</td>
<td><strong>pop</strong> removes the item at the top of the stack</td>
</tr>
<tr>
<td>push 1</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Abstract Data Types

Example

User
push 8
push 3
push 7
pop ⇒ 7
pop
push 1

Stack

Operations

push
adds an item to the top of the stack

pop
removes the item at the top of the stack
Abstract Data Types

Example

User
push 8
push 3
push 7
pop ⇒ 7
pop ⇒ 3
push 1

Stack

Operations

push
adds an item to the top of the stack

pop
removes the item at the top of the stack
Abstract Data Types

Example

User
push 8
push 3
push 7
pop ⇒ 7
pop ⇒ 3
push 1

Stack

Operations

push
adds an item to the top of the stack

pop
removes the item at the top of the stack
The set of operations provided by an ADT is called the interface.

Users of an ADT only see and interact with the interface.

```
User
  create
  push 6
  push 8
  pop
  ...

Interface
  create
  destroy
  push
  pop
  ...
```
An ADT interface must:
1. clearly describe the behaviour of each operation
2. describe the conditions under which each operation can be used

Example:

**pop**
removes the item at the top of the stack
assumes that the stack is not empty
Builders of an ADT provide an implementation of its operations.

User
- create
- push 6
- push 8
- pop
- ...

Interface
- create
- destroy
- push
- pop
- ...

Implementation
```
create(...) {
  ...
}
destroy(...) {
  ...
}
push(...) {
  ...
}
pop(...) {
  ...
}
```
Users of an ADT **do not** see the implementation.

**User**
- create
- push 6
- push 8
- pop
- ...

**Interface**
- create
- destroy
- push
- pop
- ...

**Implementation**

```c
create(...) {
    ...
}

destroy(...) {
    ...
}

push(...) {
    ...
}

pop(...) {
    ...
}
```
In C, abstract data types are implemented using two files:

- A `.h` file that contains the **interface**
- A `.c` file that contains the **implementation**
The interface includes:

- **forward declaration of the struct for the concrete representation**
  - via `typedef struct t *T`
  - **the struct is not defined in the interface**
- function prototypes for all operations
- clear description of operations
  - via comments
- a contract between the ADT and clients
  - documentation describes how an operation can be used
  - and what the expected result is **as long as the operation is used correctly**
typedef struct stack *Stack;

/** Creates a new empty stack */
Stack StackNew(void);

/** Frees memory allocated to the stack */
void StackFree(Stack s);

/** Adds an item to the top of the stack */
void StackPush(Stack s, int item);

/** Removes the item at the top of the stack 
 Assumes that the stack is not empty */
int StackPop(Stack s);
The implementation includes:

- concrete definition of the data structures
  - definition of `struct t`
- function implementations for all operations
Abstract Data Types in C
Implementation — .c file

Stack.c

```c
struct stack {
    ...
};

Stack StackNew(void) {
    ...
}

void StackFree(Stack s) {
    ...
}

void StackPush(Stack s, int item) {
    ...
}

int StackPop(Stack s) {
    ...
}
```
A user of an ADT \textit{includes} the interface and uses the interface functions to interact with the ADT.

```c
#include "Stack.h"

int main(void) {
    Stack s = StackNew();
    StackPush(s, 6);
    StackPush(s, 8);
    int item = StackPop(s);
    ...
}
```
Users of an ADT only see and interact with the interface — they do not see the implementation!

```
#include "Stack.h"

int main(void) {
    Stack s = StackNew();
    StackPush(s, 6);
    StackPush(s, 8);
    int item = StackPop(s);
    ...
}
```

```
typedef struct stack *Stack;
...
```

```
struct stack {
    ...
};
```
Abstract Data Types in C

Users of an ADT only see and interact with the interface — they do not see the implementation!

```
#include "Stack.h"

int main(void) {
    Stack s = StackNew();
    // this is not valid!
    s->...
}
```

```
#include "Stack.h"

typedef struct stack *Stack;
...
```

```
struct stack {
    ...
};
```

This means users cannot access the concrete representation (struct) directly.
Naming conventions:

- ADTs are defined in files whose names start with an uppercase letter
  - For example, for a Stack ADT:
    - The interface is defined in `Stack.h`
    - The implementation is defined in `Stack.c`
- ADT interface function names are in PascalCase and begin with the name of the ADT
Creating/Using Abstract Data Types

1. Decide what operations you want to provide
   - Operations to create, query, manipulate
   - What are their inputs and outputs?
   - What are the conditions under which they can be used (if any)?

2. Provide the function signatures and documentation for these operations in a .h file

3. The “developer” builds a concrete implementation for the ADT in a .c file

4. The “user” #includes the interface in their program and uses the provided functions
What operations can you perform on a simple bank account?

- Open an account
- Check balance
- Deposit money
- Withdraw money
typedef struct account *Account;

/** Opens a new account with zero balance */
Account AccountOpen(void);

/** Closes an account */
void AccountClose(Account acc);

/** Returns account balance */
int AccountBalance(Account acc);

/** Withdraws money from account
   Returns true if enough balance, false otherwise
   Assumes amount is positive */
bool AccountWithdraw(Account acc, int amount);

/** Deposits money into account
   Assumes amount is positive */
void AccountDeposit(Account acc, int amount);
```c
int main(void) {
    Account acc = AccountOpen();
    printf("Balance: %d\n", AccountBalance(acc));

    AccountDeposit(acc, 50);
    printf("Balance: %d\n", AccountBalance(acc));

    AccountWithdraw(acc, 20);
    printf("Balance: %d\n", AccountBalance(acc));

    AccountWithdraw(acc, 40);
    printf("Balance: %d\n", AccountBalance(acc));

    AccountClose(acc);
}
```
Invalid usage of an ADT (breaking abstraction):

```c
int main(void) {
    Account acc = AccountOpen();

    acc->balance = 1000000;

    // I'm a millionaire now, woohoo!
    printf("Balance: %d\n", AccountBalance(acc));

    AccountClose(acc);
}
```
Examples of ADTs

- Stack
- Queue
- Set
- Multiset
- Map
- Graph
- Priority Queue
A stack is a collection of items, such that the last item to enter is the first item to leave:

**Last In, First Out (LIFO)**

(Think stacks of books, plates, etc.)
A stack is a collection of items, such that the last item to enter is the first item to leave:

**Last In, First Out (LIFO)**

(Think stacks of books, plates, etc.)

- web browser history
- text editor undo/redo
- balanced bracket checking
- HTML tag matching
- RPN calculators
  (...and programming languages!)
- function calls
A stack supports the following operations:

- **push**
  - add a new item to the top of the stack

- **pop**
  - remove the topmost item from the stack

- **size**
  - return the number of items on the stack

- **peek**
  - get the topmost item on the stack without removing it
A Stack ADT can be used to check for balanced brackets.

Example of balanced brackets:
( [ { } ] )

Examples of unbalanced brackets!
( ) ) ) ( ( ( [ [ { } ] ] ( [ ] ) ( [ ] )
Stack ADTs

Example: Balancing Brackets

Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
</tbody>
</table>
Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>( [</td>
<td>-</td>
</tr>
</tbody>
</table>
Stack ADTs

Example: Balancing Brackets

Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>( [</td>
<td>-</td>
</tr>
<tr>
<td>{</td>
<td>( [ {</td>
<td>-</td>
</tr>
</tbody>
</table>
Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>( [</td>
<td>-</td>
</tr>
<tr>
<td>{</td>
<td>( [ {</td>
<td>-</td>
</tr>
<tr>
<td>}</td>
<td>( [</td>
<td>{ = }</td>
</tr>
</tbody>
</table>
Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>( [</td>
<td>-</td>
</tr>
<tr>
<td>{</td>
<td>( [ {</td>
<td>-</td>
</tr>
<tr>
<td>}</td>
<td>( [</td>
<td>{ = }</td>
</tr>
<tr>
<td>]</td>
<td>(</td>
<td>[ = ]</td>
</tr>
</tbody>
</table>
Stack ADTs

Example: Balancing Brackets

Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>]</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>{</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>}</td>
<td>(</td>
<td>{ = }</td>
</tr>
<tr>
<td>]</td>
<td>(</td>
<td>[ = ]</td>
</tr>
<tr>
<td>)</td>
<td></td>
<td>( = )</td>
</tr>
</tbody>
</table>
### Stack ADTs

**Example: Balancing Brackets**

#### Sample input:

```
( [ { } ] )
```

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>{</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>}</td>
<td>(</td>
<td>{ = }</td>
</tr>
<tr>
<td>]</td>
<td>(</td>
<td>[ = ]</td>
</tr>
<tr>
<td>)</td>
<td>(</td>
<td>( = )</td>
</tr>
<tr>
<td>EOF</td>
<td>is empty</td>
<td></td>
</tr>
</tbody>
</table>
Stack ADTs

Example: Balancing Brackets

Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>(</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
Stack ADTs

Example: Balancing Brackets

Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>(</td>
<td>-</td>
</tr>
</tbody>
</table>
### Stack ADTs

**Example Usage**

Interface

Implementation

---

**Sample input:** ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>]</td>
<td>( [</td>
<td>-</td>
</tr>
<tr>
<td>{</td>
<td>( [ {</td>
<td>-</td>
</tr>
</tbody>
</table>
Stack ADTs

Example: Balancing Brackets

Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>]</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>{</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>}</td>
<td>(</td>
<td>{ = }</td>
</tr>
</tbody>
</table>
Stack ADTs
Example: Balancing Brackets

Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>[</td>
<td>( [</td>
<td>-</td>
</tr>
<tr>
<td>{</td>
<td>( [ {</td>
<td>-</td>
</tr>
<tr>
<td>}</td>
<td>( [</td>
<td>{ = }</td>
</tr>
<tr>
<td>)</td>
<td>(</td>
<td>[ ≠ ]</td>
</tr>
</tbody>
</table>
Sample input: ( [ { } ] )

<table>
<thead>
<tr>
<th>char</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>)</td>
<td></td>
<td>fail!</td>
</tr>
<tr>
<td>[</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>(</td>
<td>-</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
typedef struct stack *Stack;

/** Creates a new, empty Stack */
Stack StackNew(void);

/** Frees memory allocated for a Stack */
void StackFree(Stack s);

/** Adds an item to the top of a Stack */
void StackPush(Stack s, Item it);

/** Removes an item from the top of a Stack
   Assumes that the Stack is not empty */
Item StackPop(Stack s);

/** Gets the number of items in a Stack */
int StackSize(Stack s);

/** Gets the item at the top of a Stack
   Assumes that the Stack is not empty */
Item StackPeek(Stack s);
How to implement a stack?

- array
- linked list
Dynamically allocate an array with an initial capacity

Fill the array sequentially — s[0], s[1], ...

Maintain a counter of the number of items on the stack
Stack ADT
Array implementation

User's view

Concrete representation

struct stack
items
size
capacity

Stack

4 6 7 3

User's view of a stack with items 4, 6, 7, and 3, and a capacity of 8.

Concrete representation of a stack with items 4, 6, 7, and 3.
Example

Perform the following operations:

PUSH(9), PUSH(2), PUSH(6), POP, POP, PUSH(8)
Stack ADT
Array implementation

PUSH(9)  PUSH(2)  PUSH(6)  POP  POP  PUSH(8)

User's view

Concrete representation

struct stack

items
size 0
capacity 8
Stack

9 2 6 8
Stack ADT
Array implementation

Push(9)  Push(2)  Push(6)  Pop  Pop  Push(8)

User’s view

Concrete representation

```c
struct stack
{
    int items[8];
    int size;
    int capacity;
}

Stack
```
Stack ADT
Array implementation

Abstraction
ADTs
Stacks
Example Usage
Interface
Implementation
Array
Linked list
Queues
Sets

PUSH(9)  PUSH(2)  PUSH(6)  POP  POP  PUSH(8)

User’s view
Concrete representation

struct stack
items
size
capacity

9 2
2
8

2 9

Stack

pUser’s view
Concrete representation
Stack ADT
Array implementation

PUSH(9)  PUSH(2)  PUSH(6)  POP  POP  PUSH(8)

User's view

Concrete representation

```
struct stack
    size = 3
    capacity = 8
    items = [9, 2, 6]
```

Stack
Stack ADT
Array implementation

**Example Usage**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH(9)</td>
<td>Add 9 to the stack</td>
</tr>
<tr>
<td>PUSH(2)</td>
<td>Add 2 to the stack</td>
</tr>
<tr>
<td>PUSH(6)</td>
<td>Add 6 to the stack</td>
</tr>
<tr>
<td>POP</td>
<td>Remove the top item (6)</td>
</tr>
<tr>
<td>POP</td>
<td>Remove the top item (2)</td>
</tr>
<tr>
<td>PUSH(8)</td>
<td>Add 8 to the stack</td>
</tr>
</tbody>
</table>

**User's view**

- push(9)
- push(2)
- push(6)
- pop ⇒ 6
- pop
- push(8)

**Concrete representation**

- struct stack
  - items: [9, 2]
  - size: 2
  - capacity: 8

- Stack
Stack ADT
Array implementation

PUSH(9)  PUSH(2)  PUSH(6)  POP ⇒ 6  POP ⇒ 2  PUSH(8)

User’s view

Concrete representation

struct stack

9

items

size 1

capacity 8

Stack
Stack ADT
Array implementation

PUSH(9)  PUSH(2)  PUSH(6)  POP ⇒ 6  POP ⇒ 2  PUSH(8)

User’s view

Concrete representation

struct stack

items
size
capacity

Stack
Cost of push:

- Inserting item at index $\text{size}$ is $O(1)$
- What if array is full?
  - If we double the size of the array with $\text{realloc(3)}$ each time it is full, push will still be $O(1)$ on average

Cost of pop:

- Accessing item at index $(\text{size} - 1)$ is $O(1)$
Stack ADT

Linked list implementation

Store items in a linked list

To push an item, insert it at the beginning of the list

To pop an item, remove it from the beginning of the list
Stack ADT
Linked list implementation

Abstraction
ADTs
Stacks
Example Usage
Interface
Implementation
Array
Linked list

Queues
Sets

User’s view
Concrete representation

struct stack

items

size

Stack
Example

Perform the following operations:

PUSH(9), PUSH(2), PUSH(6), POP, POP, PUSH(8)
Stack ADT
Linked list implementation

PUSH(9)  PUSH(2)  PUSH(6)  POP  POP  PUSH(8)

User's view

Concrete representation

struct stack
  size: 0
  items: NULL

Stack
**Stack ADT**

**Linked list implementation**

### Example Usage

<table>
<thead>
<tr>
<th>Operation</th>
<th>User's View</th>
<th>Concrete Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH(9)</td>
<td></td>
<td><img src="#" alt="Stack Diagram" /></td>
</tr>
<tr>
<td>PUSH(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH(6)</td>
<td><img src="#" alt="Stack Diagram" /></td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td><img src="#" alt="Stack Diagram" /></td>
<td>6</td>
</tr>
<tr>
<td>POP</td>
<td><img src="#" alt="Stack Diagram" /></td>
<td>2</td>
</tr>
<tr>
<td>PUSH(8)</td>
<td><img src="#" alt="Stack Diagram" /></td>
<td></td>
</tr>
</tbody>
</table>

**Structure of Stack**

```c
struct stack
{
    int size;
    int* items;
}
```

**User's View**

- **PUSH(9)**
- **PUSH(2)**
- **PUSH(6)**
- **POP**
- **POP**
- **PUSH(8)**
Stack ADT

Linked list implementation

- **PUSH(9)**
- **PUSH(2)**
- **PUSH(6)**
- **POP**
- **POP**
- **PUSH(8)**

**User’s view**

```
struct stack {
  int size;
  int items[ ];
  Stack
}
```

**Concrete representation**

```
push(9)
push(2)
push(6)
pop  ⇒ 6
pop  ⇒ 2
push(8)
```
Abstraction
ADTs
Stacks
Example Usage
Interface
Implementation
Array
Linked list
Queues
Sets

**Stack ADT**

Linked list implementation

### Example Usage

#### Interface

- `push(i)`
- `pop()`

#### Implementation

**Array**

- `push(9)`
- `push(2)`
- `push(6)`
- `pop` ➔ 6
- `pop` ➔ 2
- `push(8)`

**Linked list**

- `struct stack`
- `size` ➔ 3
- `items` ➔ NULL
- Stack

**User’s view**

**Concrete representation**
Stack ADT
Linked list implementation

Push(9) Push(2) Push(6) Pop ⇔ 6 Pop Push(8)

User's view

Concrete representation

struct stack

<table>
<thead>
<tr>
<th>items</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

pUser's view

Concrete representation
Stack ADT
Linked list implementation

Example Usage

\[
\begin{align*}
\text{PUSH}(9) & \quad \text{PUSH}(2) & \quad \text{PUSH}(6) & \quad \text{POP} \Rightarrow 6 & \quad \text{POP} \Rightarrow 2 & \quad \text{PUSH}(8)
\end{align*}
\]

User’s view

Concrete representation
Stack ADT

Linked list implementation

PUSH(9)  PUSH(2)  PUSH(6)  POP ⇒ 6  POP ⇒ 2  PUSH(8)

User’s view

Concrete representation

struct stack

items

size

Stack
Cost of push:

- Inserting at the beginning of a linked list is $O(1)$

Cost of pop:

- Removing from the beginning of a linked list is $O(1)$
A queue is a collection of items, such that the first item to enter is the first item to leave:

**First In, First Out (FIFO)**

(Think queues of people, etc.)
A **queue** is a collection of items, such that the **first** item to enter is the **first** item to leave: **First In, First Out (FIFO)**

(Think queues of people, etc.)

- waiting lists
- call centres
- access to shared resources (e.g., printers)
- processes in a computer
A queue supports the following operations:

enqueue
add a new item to the end of the queue

decqueue
remove the item at the front of the queue

size
return the number of items in the queue

peek
get the frontmost item of the queue, without removing it
typedef struct queue *Queue;

/** Create a new, empty Queue */
Queue QueueNew(void);

/** Free memory allocated to a Queue */
void QueueFree(Queue q);

/** Add an item to the end of a Queue */
void QueueEnqueue(Queue q, Item it);

/** Remove an item from the front of a Queue
 Assumes that the Queue is not empty */
Item QueueDequeue(Queue q);

/** Get the number of items in a Queue */
int QueueSize(Queue q);

/** Get the item at the front of a Queue
 Assumes that the Queue is not empty */
Item QueuePeek(Queue q);
Queue ADT

Implementation

How to implement a queue?

array

linked list (easier)
Queue ADT
Linked list implementation

To enqueue an item, insert it at the end of the list

To dequeue an item, remove it from the beginning of the list
Queue ADT
Linked list implementation

What’s the problem with this design?

User’s view
Concrete representation
Queue ADT
Linked list implementation

Improved design

User's view

Concrete representation

struct queue

front

back

size

4

Queue

front

back

size

4

4 → 6 → 7 → 3

3
Example

Perform the following operations:

ENQ(9), ENQ(2), ENQ(6), DEQ, DEQ, ENQ(8)
Queue ADT
Linked list implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ  DEQ  ENQ(8)

User’s view

Concrete representation

struct queue

front
NULL

back
NULL

size
0

Queue
Queue ADT
Linked list implementation

**User’s view**

- `enQ(9)`
- `enQ(2)`
- `enQ(6)`
- `deQ` → `9`
- `deQ` → `2`
- `enQ(8)`

**Concrete representation**

- Front: 9
- Back: 2
- Size: 1

```
struct queue
{
    int front;
    int back;
    int size;
};
```

- Queue:
  - Front: 9
  - Back: 2
  - Size: 1

---

Abstraction
ADTs
Stacks
Queues
Interface
Implementation
Linked list
Array
Sets
Queue ADT
Linked list implementation

Queue ADT
Linked list implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ  DEQ  ENQ(8)

User's view
Concrete representation
Queue ADT
Linked list implementation

EnQ(9)  EnQ(2)  EnQ(6)  DeQ  DeQ  EnQ(8)

User’s view  Concrete representation

front  back

struct queue

front
back
size
Queue

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

front = NULL, back = NULL

size = 3
Queue ADT
Linked list implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ ⇒ 9  DEQ  ENQ(8)

User's view

Concrete representation
Queue ADT
Linked list implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ ⇒ 9  DEQ ⇒ 2  ENQ(8)

User’s view

Concrete representation

struct queue

front  back  size

Queue
Queue ADT
Linked list implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ ⇒ 9  DEQ ⇒ 2  ENQ(8)

User’s view
Concrete representation
Cost of enqueue:
  • Inserting at the end of the linked list is $O(1)$

Cost of dequeue:
  • Removing from the beginning of the linked list is $O(1)$
Queue ADT
Array implementation

Dynamically allocate an array with an initial capacity
Maintain an index to the front of the queue
Maintain a counter of the number of items in the queue
Queue ADT
Array implementation

User’s view

Concrete representation

struct queue

back

front

3
7
6
4

Queue

front
0

capacity
8

size
4

items

4 6 7 3
Example

Perform the following operations:

$\text{ENQ}(9), \text{ENQ}(2), \text{ENQ}(6), \text{DEQ}, \text{DEQ}, \text{ENQ}(8)$
Queue ADT
Array implementation

User's view

Concrete representation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ  DEQ  ENQ(8)

front
back

struct queue

items
size 0
capacity 8
front 0

Queue

pUser's viewp

Concrete representation
Queue ADT

Array implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ  DEQ  ENQ(8)

User’s view

Concrete representation

struct queue

front 0

back

items

size 1

capacity 8

Queue
Queue ADT
Array implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ  DEQ  ENQ(8)

User's view

Concrete representation

front
9
2
back

struct queue
items
size
2
capacity
8
front
0
Queue
Queue ADT
Array implementation

### User's view

<table>
<thead>
<tr>
<th>enqueue</th>
<th>dequeue</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENQ(9)</td>
<td>9</td>
</tr>
<tr>
<td>ENQ(2)</td>
<td>2</td>
</tr>
<tr>
<td>ENQ(6)</td>
<td>6</td>
</tr>
<tr>
<td>DEQ</td>
<td></td>
</tr>
<tr>
<td>DEQ</td>
<td></td>
</tr>
<tr>
<td>ENQ(8)</td>
<td></td>
</tr>
</tbody>
</table>

### Concrete representation

```
struct queue
{
    front, back, items, size, capacity

    front = 0
    size = 3
    capacity = 8
    items = [9, 2, 6]
}
```
Queue ADT
Array implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ ⇒ 9  DEQ  ENQ(8)

User’s view

Concrete representation

struct queue

items
size
capacity
front

2  6
2
8
1

Queue
Queue ADT
Array implementation

ENQ(9)  ENQ(2)  ENQ(6)  DEQ ⇒ 9  DEQ ⇒ 2  ENQ(8)

User’s view

Concrete representation

struct queue

front

6

size

1

capacity

8

items

DEQ ⇒ 9

DEQ ⇒ 2

enQ(9)
enQ(2)
enQ(6)
deQ

deQ

back

Queue
Queue ADT
Array implementation

ENQ(9)   ENQ(2)   ENQ(6)   DEQ ⇒ 9   DEQ ⇒ 2   ENQ(8)

User’s view

Concrete representation

struct queue

front: 2

size: 2

capacity: 8

items: [9, 2, 6, 2, 8]

back

Queue
Cost of enqueue:

• Dequeue involves calculating insertion index and inserting item at that index $\Rightarrow O(1)$

Cost of dequeue:

• Dequeue involves accessing item at index front $\Rightarrow O(1)$
A set is an unordered collection of distinct elements. In this lecture we are concerned with sets of integers.
Basic set operations:

- Create an empty set
- Insert an item into the set
- Delete an item from the set
- Check if an item is in the set
- Get the size of the set
- Display the set
#include <stdbool.h>

typedef struct set *Set;

/** Creates a new empty set */
Set SetNew(void);

/** Free memory used by set */
void SetFree(Set set);

/** Inserts an item into the set */
void SetInsert(Set set, int item);

/** Deletes an item from the set */
void SetDelete(Set set, int item);

/** Checks if an item is in the set */
bool SetContains(Set set, int item);

/** Returns the size of the set */
int SetSize(Set set);

/** Displays the set */
void SetShow(Set set);
Counting and displaying distinct numbers:

```c
#include <stdio.h>
#include "Set.h"

int main(void) {
    Set s = SetNew();

    int val;
    while (scanf("%d", &val) == 1) {
        SetInsert(s, val);
    }

    printf("Number of distinct values: %d\n", SetSize(s));
    printf("Values: ");
    SetShow(s);

    SetFree(s);
}
```
Different ways to implement a set:

- Unordered array
- Ordered array
- Ordered linked list
Set Implementation

Unordered array

```
struct set {
    int capacity;  // 8
    int size;     // 5
    int* elems;   // 4, 7, 5, 1, 9
}
```

- `capacity`: 8
- `size`: 5
- `elems`: [4, 7, 5, 1, 9]
How do we check if an element exists?

- Perform linear scan of array \( \Rightarrow O(n) \)

```c
bool SetContains(Set s, int elem) {
    for (int i = 0; i < s->size; i++) {
        if (s->elems[i] == elem) {
            return true;
        }
    }
    return false;
}
```
How do we insert an element?

- If the element doesn’t exist, insert it after the last element

```c
void SetInsert(Set s, int elem) {
    if (SetContains(s, elem)) {
        return;
    }
    if (s->size == s->capacity) {
        // error message
    }
    s->elems[s->size] = elem;
    s->size++;
}
```

Time complexity: $O(n)$

- SetContains is $O(n)$ and inserting after the last element is $O(1)$
Abstraction
ADTs
Stacks
Queues
Sets
Interface
Example Usage
Implementation
Unordered array
Ordered array
Linked list
Summary

How do we delete an element?

• If the element exists, overwrite it with the last element

```c
void SetDelete(Set s, int elem) {
    for (int i = 0; i < s->size; i++) {
        if (s->elems[i] == elem) {
            s->elems[i] = s->elems[s->size - 1];
            s->size--;
            return;
        }
    }
}
```

Time complexity: $O(n)$

• Finding the element is $O(n)$, overwriting it with the last element is $O(1)$
Set Implementation

Ordered array

```
struct set {
  int capacity; // 8
  int size;    // 5
  int elems[5]; // 1, 4, 5, 7, 9
}
```

Set
How do we check if an element exists?

- Perform binary search $\Rightarrow O(\log n)$

```c
def SetContains(Set s, int elem) {
    int lo = 0;
    int hi = s->size - 1;

    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (elem < s->elems[mid]) {
            hi = mid - 1;
        } else if (elem > s->elems[mid]) {
            lo = mid + 1;
        } else {
            return true;
        }
    }
    return false;
}
```
Set Implementation

Ordered array

How do we insert an element?

• Use binary search to find the index of the smallest element which is *greater than or equal to* the given element

• If this element *is* the given element, then it already exists, so no need to do anything

• Otherwise, insert the element at that index and shift everything greater than it up
Time complexity of insertion?

- Binary search lets us find the insertion point in $O(\log n)$ time
- ...but we still have to potentially shift up to $n$ elements, which is $O(n)$
Set Implementation
Ordered array

How do we delete an element?

- Use binary search to find the element
- If the element exists, shift everything greater than it down

Time complexity?

- Binary search lets us find the element in $O(\log n)$ time
- ...but we still have to potentially shift up to $n$ elements, which is $O(n)$
Set Implementation
Ordered linked list

struct set
  size 3
  elems
    4
    5
    7

Summary

Sets
  Interface
  Example Usage
  Implementation
  Unordered array
  Ordered array
  Linked list

Example Usage

Implementation

Unordered array

Ordered array
How do we check if an element exists?

- Traverse the list $\Rightarrow O(n)$

```c
bool SetContains(Set s, int elem) {
    for (struct node *curr = s->elems; curr != NULL; curr = curr->next) {
        if (curr->elem == elem) {
            return true;
        }
    }
    return false;
}
```
We always have to traverse the list from the start. Therefore...

- Insertion and deletion are also $O(n)$

However, this analysis hides a crucial advantage of linked lists:

- Finding the insertion/deletion point is $O(n)$
- But inserting/deleting a node is $O(1)$, as no shifting is required
### Set ADT Summary

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Contains</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered array</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Ordered array</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Ordered linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
https://forms.office.com/r/5c0fb4tvMb