COMP2521 24T2
Abstract Data Types

Sim Mautner
cs2521@cse.unsw.edu.au

abstraction
abstract data types
stacks and queues
sets

Slides adapted from those by Kevin Luxa 2521 24T1
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
1: 48 89 e5
4: 89 7d ec
7: c7 45 fc 01 00 00 00
e: c7 45 f8 01 00 00 00
15: eb 0e
17: 8b 45 fc
1a: 0f af 45 f8
1e: 89 45 fc
21: 83 45 f8 01
25: 8b 45 f8
28: 3b 45 ec
2b: 7e ea
2d: 8b 45 fc
30: 5d
31: c3
```
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.
Abstract Data Types
Example - List

• What makes a List, a List?
• What things do we need a List to do?
  • create
  • add to start
  • add to end
  • remove from start
  • remove from end
  • destroy
The set of operations provided by an ADT is called the **interface**.

Users of an ADT only see and interact with the interface.
An ADT interface must:
1. clearly describe the behaviour of each operation
2. describe the conditions under which each operation can be used

Example:

**remove from end**
removes the item at the end of the list
assumes that the list is not empty
Abstract Data Types
Example - List

- What makes a List, a List?
- What things do we need a List to do?
- What details do we need to know before we can use the List?
  - https://cgi.cse.unsw.edu.au/~cs2521/24T2/lectures/code/wk3-adts/list-adt/List.h
- What would it look like to use the List?
  - https://cgi.cse.unsw.edu.au/~cs2521/24T2/lectures/code/wk3-adts/list-adt/test_list.c
- What do we *not* need to know?
  - Anything else
What could be going on behind the scenes?

What are the advantages of ADTs?
What are the advantages of ADTs?

- Design by Contract: Programmers don’t need to know how the ADT will be implemented, in order to start using it. The .h file documents the agreement between both parties.
What are the advantages of ADTs?

- **Design by Contract:** Programmers don’t need to know how the ADT will be implemented, in order to start using it. The `.h` file documents the agreement between both parties.

- **Modular:** The implementation of the ADT can be changed without updates being required to the broader application.
Abstract Data Types - Benefits

What are the advantages of ADTs?

• Design by Contract: Programmers don’t need to know how the ADT will be implemented, in order to start using it. The .h file documents the agreement between both parties.

• Modular: The implementation of the ADT can be changed without updates being required to the broader application.

• Security: Prevents operations which shouldn’t be possible, from being executed on the data.
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
Abstract Data Types in C

Interface — .h file

List.h

typedef struct list List;

/** Creates a new empty list */
List * ListNew(void);

/** Frees memory allocated to the list */
void ListDestroy(List * l);

/** Adds an item to the end of the list */
void ListInsertEnd(List * l, int item);

/** Removes the item end of the list
Assumes that the list is not empty */
int ListRemoveEnd(List * l);
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 list {
    ...
};

List * ListNew(void) {
    ...
}

void ListDestroy(List * l) {
    ...
}

void ListInsertEnd(List * l, int item) {
    ...
}

int ListRemoveEnd(List * l) {
    ...
}
```
A user of an ADT includes the interface and uses the interface functions to interact with the ADT.

```c
#include "List.h"

int main(void) {
    List *l = ListNew();
    ListInsertStart(l, 6);
    ListInsertStart(l, 8);
    int item = ListRemoveEnd(l);
    ...
}
```
Naming conventions:

- ADTs are defined in files whose names start with an uppercase letter
  - For example, for a List ADT:
    - The interface is defined in List.h
    - The implementation is defined in List.c
  
- ADT interface function names are in PascalCase and begin with the name of the ADT
Examples of ADTs

- Stack
- Queue
- Set
- Multiset
- Map
- Graph
- Priority Queue
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: Stack

User

Stack

Operations

push
adds an item to the top of the stack

pop
removes the item at the top of the 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: Stack

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: Stack

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: Stack

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: Stack

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

Stack
8
3
7
1

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: Stack

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: Stack

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: Stack

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
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)?

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

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

The “user” #includes the interface in their program and uses the provided functions
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>
<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>

eof is empty
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>

*eof is empty*
### 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>
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>
</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>
<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>
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>[ ≠ ] fail!</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

Array

Linked list

Queues

Sets
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
Stack ADT

Array implementation

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

User’s view

Concrete representation

struct stack

- items
- size: 1
- capacity: 8

Stack
Stack ADT
Array implementation

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

User's view

Concrete representation

struct stack

items
size
capacity

Stack
Stack ADT
Array implementation

Example Usage

\[\text{push}(9) \quad \text{push}(2) \quad \text{push}(6) \quad \text{pop} \quad \text{pop} \quad \text{push}(8)\]

User's view

Concrete representation
Stack ADT
Array implementation

User’s view

Concrete representation

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

Stack

struct stack

items

size

capacity

9 2

2

8
Stack ADT
Array implementation

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

**User’s view**

**Concrete representation**
Abstraction

ADTs

Stacks

Example Usage

Interface

Implementation

Array

Linked list

Queues

Sets

Stack ADT

Array implementation

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

User’s view

Concrete representation
Cost of push:
- Inserting item at index size is $O(1)$
- What if array is full?
  - If we double the size of the array with `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

User’s view

Concrete representation

struct stack

items

size

Stack

Array

Linked list

Example Usage

Interface

Implementation

Queues

Sets

Abstraction

ADTs

Stacks

Sets

Example Usage

Interface

Implementation

Array

Linked list
Example

Perform the following operations:

\textbf{PUSH}(9), \textbf{PUSH}(2), \textbf{PUSH}(6), \textbf{POP}, \textbf{POP}, \textbf{PUSH}(8)
Stack ADT

Linked list implementation

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

User’s view

Concrete representation

struct stack

items | NULL
size  | 0

Stack

pUser's view

Concrete representation
Stack ADT

Linked list implementation

**Example Usage**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH(9)</td>
<td>9</td>
</tr>
<tr>
<td>PUSH(2)</td>
<td>2</td>
</tr>
<tr>
<td>PUSH(6)</td>
<td>6</td>
</tr>
<tr>
<td>POP</td>
<td>6</td>
</tr>
<tr>
<td>POP</td>
<td>2</td>
</tr>
<tr>
<td>PUSH(8)</td>
<td>8</td>
</tr>
</tbody>
</table>

**Concrete representation**

- **User's view**
  - Stack
    - `size`: 2
    - `items`: 9, 2, 9

- **Concrete representation**
  - `struct stack`
    - `size`: 2
    - `items`: 9, 2, 9

**Interface**

- `push(x)`
- `pop()`

**Implementation**

- Linked list implementation
Stack ADT

Linked list implementation

User’s view

Concrete representation

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

<table>
<thead>
<tr>
<th>6</th>
<th>2</th>
<th>9</th>
</tr>
</thead>
</table>

struct stack

- items
- size

Stack

Example Usage

Abstraction

ADTs

Stacks

Queues

Sets
Stack ADT
Linked list implementation

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

User’s view

Concrete representation

struct stack

items

size

Stack
Stack ADT

Linked list implementation

User's view

Concrete representation

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

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

User’s view

Concrete representation
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
- **dequeue**: 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);
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
What’s the problem with this design?

User’s view

Concrete representation
Queue ADT
Linked list implementation

Improved design

User's view
Concrete representation
Example

Perform the following operations:

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

User's view

Concrete representation

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

struct queue

front NULL
back NULL
size 0

Queue
Queue ADT
Linked list implementation

<table>
<thead>
<tr>
<th>ENQ(9)</th>
<th>ENQ(2)</th>
<th>ENQ(6)</th>
<th>DEQ</th>
<th>DEQ</th>
<th>ENQ(8)</th>
</tr>
</thead>
</table>

User's view

Concrete representation

struct queue

front

back

size

Queue
Queue ADT
Linked list implementation

<table>
<thead>
<tr>
<th>ENQ(9)</th>
<th>ENQ(2)</th>
<th>ENQ(6)</th>
<th>DEQ</th>
<th>DEQ</th>
<th>ENQ(8)</th>
</tr>
</thead>
</table>

User's view

Concrete representation

struct queue

front
back
size

Queue
Queue ADT

Linked list implementation

User’s view

Concrete representation

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

front
back

struct queue

size

Queue

pUser's view
Queue ADT
Linked list implementation

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

User’s view

Concrete representation

struct queue

front

back

size 2

Queue

2

6

front

back

size 2

Queue
Queue ADT
Linked list implementation

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

User’s view

Concrete representation

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

Queue
back
front

Linked list implementation

<table>
<thead>
<tr>
<th>enQ(9)</th>
<th>enQ(2)</th>
<th>enQ(6)</th>
<th>deQ ⇒ 9</th>
<th>deQ ⇒ 2</th>
<th>enQ(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>
Queue ADT
Linked list implementation

\begin{align*}
\text{ENQ}(9) & \quad \text{ENQ}(2) & \quad \text{ENQ}(6) & \quad \text{DEQ} \Rightarrow 9 & \quad \text{DEQ} \Rightarrow 2 & \quad \text{ENQ}(8)
\end{align*}

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

front 0

capacity 8

size 4

items

4 6 7 3
Example

Perform the following operations:

```
enQ(9), enQ(2), enQ(6), deQ, deQ, enQ(8)
```
Queue ADT
Array implementation

User’s view vs Concrete representation

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

struct queue
items
size
capacity
front

Queue
Queue ADT
Array implementation

User’s view

Concrete representation

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

Struct queue:
- front: 0
- back: 9
- size: 1
- capacity: 8
- items: 9, 2, 6

Queue:
- front: 0
- back: 9
- size: 1
- capacity: 8
- items: 9, 2, 6, 8
Queue ADT
Array implementation

User's view

Concrete representation

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

<table>
<thead>
<tr>
<th>front</th>
<th>back</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

struct queue

- items: 2
- size: 2
- capacity: 8
- front: 0

Queue
**Queue ADT**

Array implementation

The diagram illustrates a queue with the following operations:

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

The queue is represented with a `struct queue` that includes:

- **items**
- **size**
- **capacity**
- **front**

In the `struct queue`:

- **front** = 0
- **size** = 3
- **capacity** = 8
- **items**:
  - 9
  - 2
  - 6

The queue's back is indicated by the number 9, as it is the last element added.

**User's view** vs **Concrete representation**
Queue ADT
Array implementation

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

User’s view

Concrete representation

struct queue
  front 1
  capacity 8
  size 2
  items
  2 6
  back

Queue ADT
Array implementation

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

User’s view
Concrete representation

struct queue

front

size
capacity
items
back

front

Queue
Queue ADT
Array implementation

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

User’s view

Concrete representation
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
  capacity 8
  size 5
  elems
    4
    7
    5
    1
    9

Set
How do we check if an element exists?

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

```cpp
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 \(O(1)\)
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

- elems
- size: 5
- capacity: 8

Set

1 4 5 7 9
How do we check if an element exists?

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

```c
bool 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;
}
```
How do we insert an element?

- Use binary search to find the index of the smallest element which is \textit{greater than or equal to} the given element.
- If this element \textit{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 $O(n)$
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

Set
How do we check if an element exists?

- Traverse the list ⇒ $O(n)$

```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/riGKCze1cQ