COMP2521 23T3
Abstract Data Types

Kevin Luxa

cs2521@cse.unsw.edu.au

abstraction
abstract data types
stacks and queues
sets

What is abstraction?

Abstraction

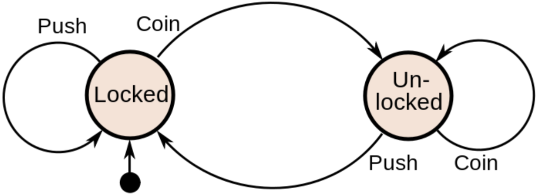
is the process of
hiding or generalising
the details of an object or system
to focus on its high-level meaning or behaviour

Using an `int` or `double` in C

Writing a function

Writing a program in C instead of assembly

Modelling the states of a system using a state machine



States of a turnstile

We drive a car by using a steering wheel and pedals

We operate a television through a remote control and on-screen display

We deposit and withdraw money to/from our bank account via an ATM

Abstraction

Why?

To use a system,
it should be enough to
understand what its components do
without knowing how...

Data Types

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 `int`s
  - operations: index lookup, index assignment

An abstract data type...

is a data type
which is described by its high-level operations
rather than how it is implemented

the set of operations provided by an ADT is called its interface

Features of ADTs:

Interface is separated from the implementation

Users of the ADT only see and interact with the interface

Builders of the ADT provide an implementation

Abstract Data Types

Benefits

Abstract data types...

facilitate decomposition

make implementation changes invisible to clients

improve readability and structuring of software

Abstract Data Types

Interface and Implementation

ADT interfaces provide

- an opaque view of a data structure
- function signatures for all operations
- semantics of operations (via documentation)
- a contract between the ADT and clients

ADT implementations provide

- a concrete definition of the data structures
- function implementations for all operations

ADTs in C

Interfaces

The interface of an ADT is defined in a `.h` file. It provides:

- an opaque view of a data structure
  - via `typedef struct t *T`
  - we do not define a concrete `struct t`
- function signatures for all operations
  - via C function prototypes
- semantics of operations (via documentation)
  - 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*

ADTs in C

Implementations

The implementation of an ADT is defined in a `.c` file. It provides:

- concrete definition of the data structures
  - definition of `struct t`
- function implementations for all operations

Conventions

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" `#include`s the interface in their program and uses the provided functions

Simple ADT Example

Bank Account

What operations can you perform on a simple bank account?

- Open an account
- Check balance
- Deposit money
- Withdraw money

Simple ADT Example

Bank Account - Interface (Account.h)

```
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);
```

Simple ADT Example

Bank Account - Usage

```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);
}
```

Simple ADT Example

Bank Account - Usage

Invalid usage of an ADT (breaking abstraction):

```
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

Stacks and Queues

Stacks and queues are

- ... ubiquitous in computing!
- ... part of many important algorithms
- ... good illustrations of ADT benefits

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.)

Stacks

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

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 constructor and a destructor
to create a new empty stack, and
to release all resources of a stack

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: `( [ { } ] )`

char	stack	check
		-
(	(	-

Sample input: ([{ }])

char	stack	check
		-
(	(	-
[	([	-

Sample input: `( [ { } ] )`

char	stack	check
		-
(	(	-
[	([	-
{	([{	-

Stack ADTs

Example: Balancing Brackets

Sample input: `( [ { } ] )`

char	stack	check
		-
(	(	-
[	([	-
{	([{	-
}	([	{ = }

Sample input: ([{ }])

char	stack	check
		-
(	(	-
[	([	-
{	([{	-
}	([	{ = }
]	(	[=]

Stack ADTs

Example: Balancing Brackets

Sample input: ([{ }])

char	stack	check
		-
(	(	-
[	([	-
{	([{	-
}	([	{ = }
]	(	[=]
)		(=)

Sample input: ([{ }])

char	stack	check
		-
(	(	-
[	([	-
{	([{	-
}	([	{ = }
]	(	[=]
)		(=)
EOF		is empty

Stack ADTs

Example: Balancing Brackets

Sample input: `( [ { } ) ]`

char	stack	check
2		-
(	(	-

Sample input: `( [ { } ) ]`

char	stack	check
2		-
(	(	-
[	([	-

Sample input: ([{ })]

char	stack	check
?		-
(	(	-
[	([	-
{	([{	-

Sample input: ([{ })]

char	stack	check
?		-
(	(	-
[	([	-
{	([{	-
}	([	{ = }

Stack ADTs

Example: Balancing Brackets

Sample input: ([{ })]

char	stack	check
2		-
(	(	-
[	([	-
{	([{	-
}	([	{ = }
)		[≠)

Sample input: `( [ { } ) ]`

char	stack	check
?		-
(	(	-
[	([	-
{	([{	-
}	([	$\{ = \}$
)		$[\neq)$
		fail!

Stack ADT Interface

"Stack.h"

```
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);
```

Stack ADTs

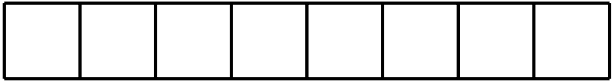
An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, ...
- Maintain a counter of the number of pushed items

Stack ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, ...
- Maintain a counter of the number of pushed items

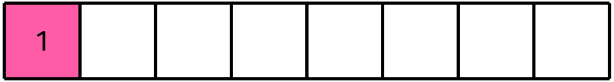


NEW

Stack ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, ...
- Maintain a counter of the number of pushed items

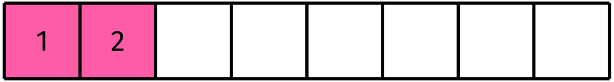


NEW   PUSH (1)

Stack ADTs

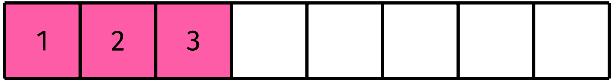
An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, ...
- Maintain a counter of the number of pushed items



NEW PUSH (1) PUSH (2)

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, ...
- Maintain a counter of the number of pushed items

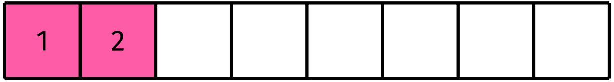


NEW PUSH (1) PUSH (2) PUSH (3)

Stack ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, ...
- Maintain a counter of the number of pushed items

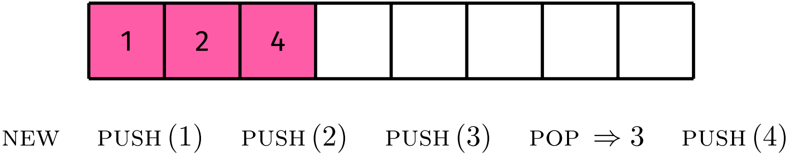


NEW   PUSH (1)   PUSH (2)   PUSH (3)   POP $\Rightarrow 3$

Stack ADTs

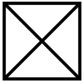
An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, ...
- Maintain a counter of the number of pushed items



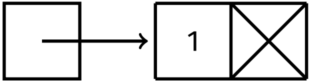
- Add node to the front of the list on push
- Take node from the front of the list on pop

- Add node to the front of the list on push
- Take node from the front of the list on pop



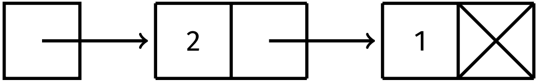
NEW

- Add node to the front of the list on push
- Take node from the front of the list on pop



NEW PUSH (1)

- Add node to the front of the list on push
- Take node from the front of the list on pop

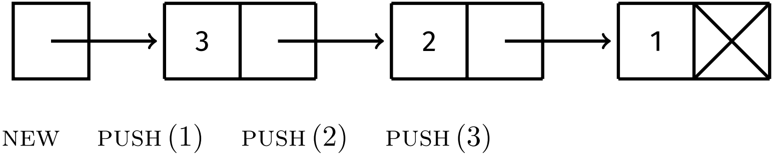


NEW PUSH (1) PUSH (2)

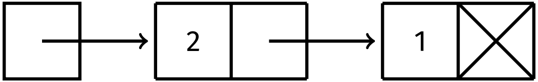
Stack ADTs

An Implementation using Linked Lists

- Add node to the front of the list on push
- Take node from the front of the list on pop



- Add node to the front of the list on push
- Take node from the front of the list on pop

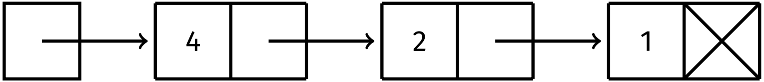


NEW PUSH (1) PUSH (2) PUSH (3) POP $\Rightarrow 3$

Stack ADTs

An Implementation using Linked Lists

- Add node to the front of the list on push
- Take node from the front of the list on pop



NEW PUSH (1) PUSH (2) PUSH (3) POP $\Rightarrow 3$ PUSH (4)

Queues

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.)

Queues

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

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

a constructor and a destructor
to create a new empty queue, and
to release all resources of a queue

Queue ADT Interface

"Queue.h"

```
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 ADTs

An Implementation using Linked Lists

We need to add and remove items from opposite ends now!

Can we do this efficiently? What do we need?

We need to add and remove items from opposite ends now!

Can we do this efficiently? What do we need?

- If we only have a pointer to the head, **no**!
  We'd need to traverse the list to the tail every time.

We need to add and remove items from opposite ends now!

Can we do this efficiently? What do we need?

- If we only have a pointer to the head, **no**!
  We'd need to traverse the list to the tail every time.
- If we have a pointer to both head *and* tail,
  we don't have to traverse, and *adding* is efficient.

Queue ADTs

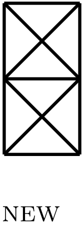
An Implementation using Linked Lists

Add nodes to the end; take nodes from the front.

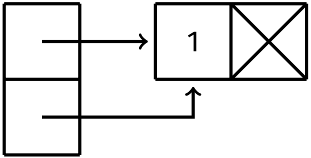
Queue ADTs

An Implementation using Linked Lists

Add nodes to the end; take nodes from the front.

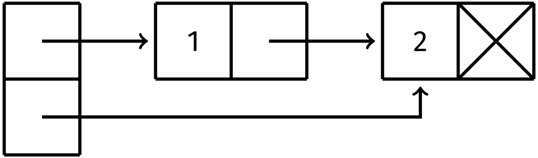


Add nodes to the end; take nodes from the front.



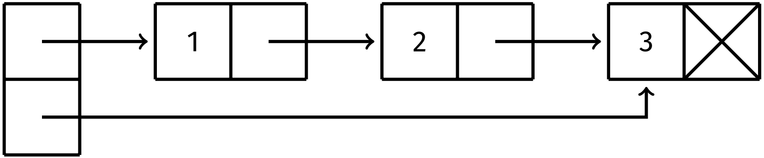
NEW ENQ (1)

Add nodes to the end; take nodes from the front.



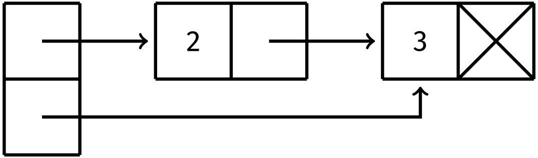
NEW ENQ (1) ENQ (2)

Add nodes to the end; take nodes from the front.



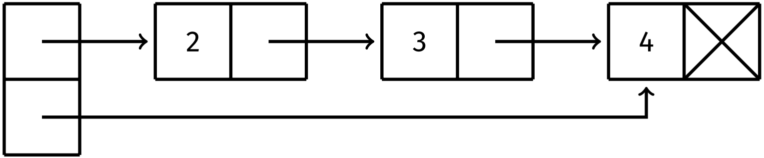
NEW   ENQ (1)   ENQ (2)   ENQ (3)

Add nodes to the end; take nodes from the front.



$\textsc{new} \quad \textsc{en}Q\,(1) \quad \textsc{en}Q\,(2) \quad \textsc{en}Q\,(3) \quad \textsc{de}Q \Rightarrow 1$

Add nodes to the end; take nodes from the front.



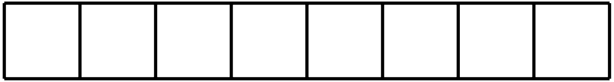
$\text{NEW} \quad \text{ENQ}\,(1) \quad \text{ENQ}\,(2) \quad \text{ENQ}\,(3) \quad \text{DEQ} \Rightarrow 1 \quad \text{ENQ}\,(4)$

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items

Queue ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items

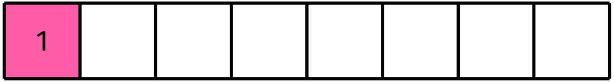


NEW

Queue ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items

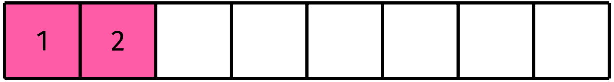


NEW ENQ (1)

Queue ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items

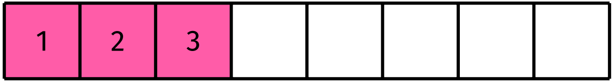


NEW ENQ (1) ENQ (2)

Queue ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items

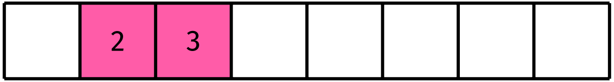


NEW ENQ (1) ENQ (2) ENQ (3)

Queue ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items

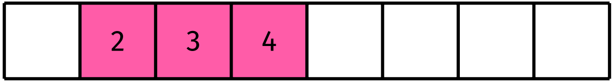


NEW ENQ (1) ENQ (2) ENQ (3) DEQ ⇒ 1

Queue ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items



NEW ENQ (1) ENQ (2) ENQ (3) DEQ $\Rightarrow 1$ ENQ (4)

Sets

A set is an unordered collection of distinct elements.

In this lecture we are concerned with sets of integers.

Set

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

Set

Interface

```
/** 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);
```

Set

Interface - "Set.h"

```
#ifndef SET_H
#define SET_H

#include <stdbool.h>

typedef struct set *Set;

// ADT function prototypes

#endif
```

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);
}
```

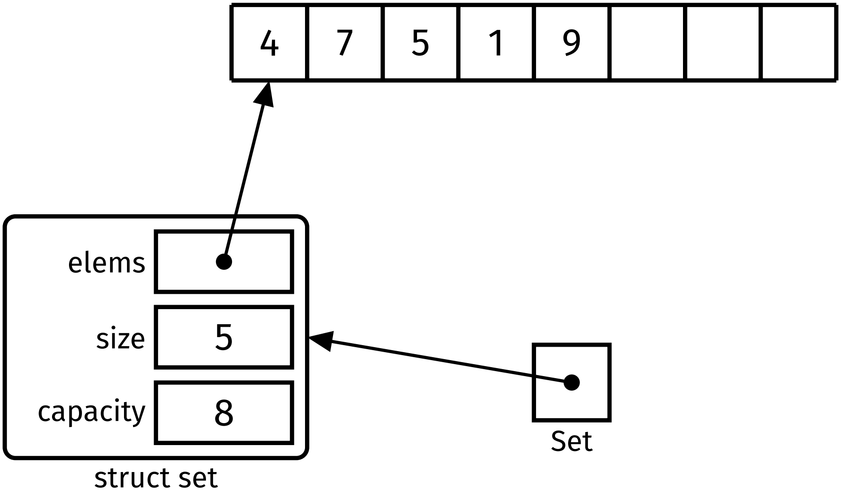
Set Implementation

Different ways to implement a set:

- Unordered array
- Ordered array
- Ordered linked list

Set Implementation

Unordered array



How do we check if an element exists?

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

```
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

```
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 $O(1)$

Set Implementation

Unordered array

How do we delete an element?

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

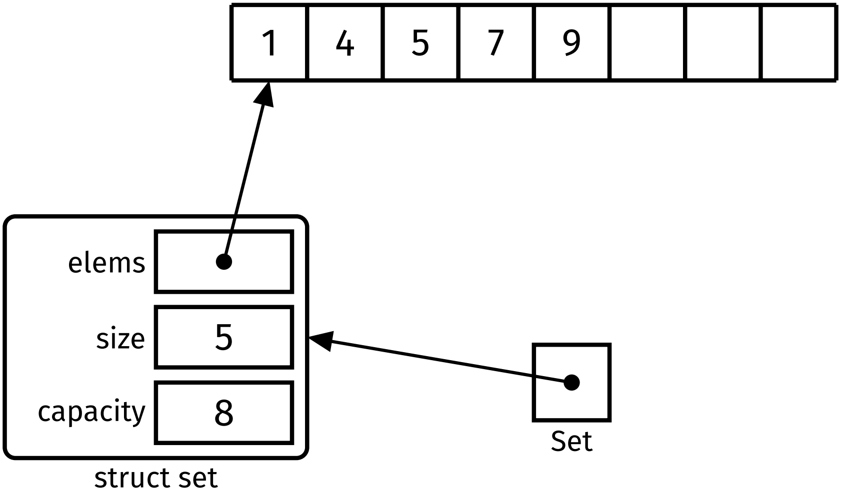
```
void SetDelete(Set s, int elem) {
    for (int i = 0; i < s->size; i++) {
        if (s->elems[i] == elem) {
            s->elems[i] = s->elems[s->size - 1];
            s->size--;
            return;
        }
    }
}
```

Time complexity: $O(n)$

- Finding the element is $O(n)$, overwriting it with the last element is $O(1)$

Set Implementation

Ordered array



How do we check if an element exists?

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

```
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 *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 $O(n)$

How do we delete an element?

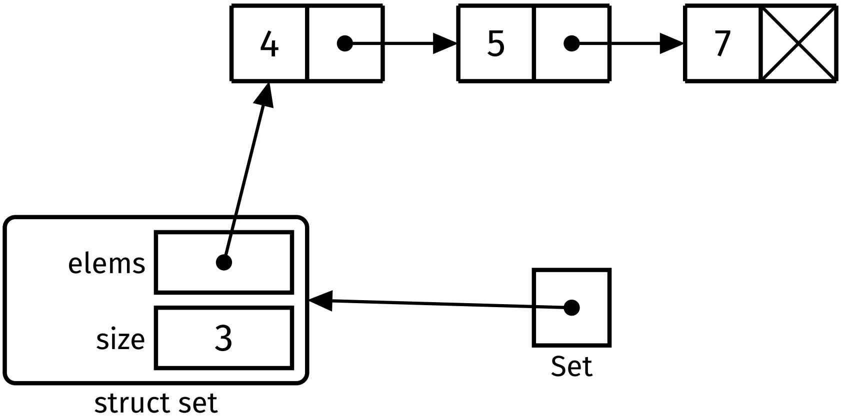
- Use binary search to find the element
- If the element exists, shift everything greater than it down

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



How do we check if an element exists?

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

```
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

Data Structure	Contains	Insert	Delete
Unordered array	$O(n)$	$O(n)$	$O(n)$
Ordered array	$O(\log n)$	$O(n)$	$O(n)$
Ordered linked list	$O(n)$	$O(n)$	$O(n)$

Feedback

https://forms.office.com/r/aPF09YHZ3X