Abstract Data Types (ADTs)

Readings: CP:AMA 19.3, 19.4, 19.5, 17.7 (`qsort`)

Selecting a data structures

Every data structure in Computer Science is either one of the following **"core"** data structures or a *combination* of two or more data structures.

- primitives (simple values such as an integer)
- structures
- arrays
- linked lists
- trees
- graphs

Selecting an appropriate data structure, and knowing how/when to combine data structures is important in program **design**.

When choosing between arrays, linked lists and BSTs, an important consideration is whether or not you need to maintain the original data ***sequence***. Linked lists and arrays can maintain the order in which elements are inserted into them, while a BST does not.

Knowing how the data will be used (*e.g.* the *frequency* of each operation or function call) and the corresponding run-time efficiencies will significantly affect your data structure selection.

Here are a few examples:

- An array is a good choice if you frequently access elements at specific positions.
- A linked list is a good choice if you frequently add/remove elements at the start of a data sequence.
- A self-balancing BST is a good choice if you frequently search for, add and remove elements from unsequenced data.

Data Structure Comparison: Sequenced Data

Function	Dyn. Array	Linked List
`item_at`	$O(1)$	$O(n)$
`search`	$O(n)$	$O(n)$
`insert_at`	$O(n)$	$O(n)$
`insert_front`	$O(n)$	$O(1)$
`insert_back`	$O(1)^*$	$O(1)^{**}$
`remove_at`	$O(n)$	$O(n)$
`remove_front`	$O(n)$	$O(1)$
`remove_back`	$O(1)$	$O(n)^{***}$

* amortized, $^{**}O(n)$ without a wrapper, $^{***}O(1)$ when *doubly* linked.

Data Structure Comparison: Sorted Data

Function	Sorted Dynamic Array	Sorted Linked List	Regular BST	Self-Balancing BST
`item_at`	$O(1)$	$O(n)$	$O(h)^*$	$O(\log n)^*$
`search`	$O(\log n)$	$O(n)$	$O(h)$	$O(\log n)$
`insert`	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$
`remove`	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$

* $O(n)$ without augmentation.

Abstract Data Types (ADTs)

Formally, an ***Abstract Data Type (ADT)*** is a mathematical model for storing and accessing data through ***operations***.

Practically, an ADT implementation is a **module** designed to store data where the underlying data structure and implementation is hidden from the client, and the client can only access the data through the interface functions (ADT operations).

One of the advantages of **modularization** is that the client only needs an *abstraction* of how the module is implemented. Hiding the implementation details from the client improves the *flexibility* and *security* of the module.

C implementation example: Point3D ADT

Consider a simple Point3D ADT that stores three coordinates and has the following operations (functions): `get_x`, `get_y`, `get_z`, `set_x`, `set_y`, `set_z`.

To implement the Point3D ADT, we may choose to store the coordinates as fields in a structure, elements in an array, or elements in a linked list. This should be hidden from the client.

A formal (mathematical) ADT may not have operations to *create* and *destroy* the ADT, but they are typically required in a practical ADT implementation: `create_Point3D`, `destroy_Point3D`.

Opaque structures

To hide the data structure from the client, we will use ***opaque structures***, where the fields of the structure are not "visible".

In C, structures can be *declared* without any fields. If a client is only provided with a declaration, only *pointers* to that type can be defined.

```
struct my_opaque_struct;          // DECLARATION
struct my_opaque_struct myvar;    // INVALID
struct my_opaque_struct *p;       // VALID
```

In full Racket, all structures are *opaque* by default. The `#:transparent` keyword is required to make fields visible.

typedef

The C `typedef` keyword allows you to create your own "type" from previously existing types. This is usually done to improve the readability of the code, or to hide the type (for security or flexibility).

```
typedef int Integer;
typedef int *IntPtr;

Integer i;
IntPtr p = &i;
```

It is common to use a different coding style (we use `CamelCase`) when defining a new "type".

`typedef` is often used to simplify complex declarations (*e.g.*: function pointer types).

```c
// point3d.h
typedef struct point3d *Point3D;

// Operations: (partial list)
Point3D create_Point3D(int x, int y, int z);
int get_x(Point3D p);
void set_x(Point3D p, int x);
```

With `typedef`, the interface and client code is clean. With opaque structures, the client cannot determine the implementation or access the data directly (security).

```c
// client.c
#include "point3d.h"

int main(void) {
  Point3D p = create_Point3D(2,3,5);
  set_x(p, 100);
  printf("%d\n", get_x(p));
}
```

Some programmers consider it poor style to "abstract" that `Point3D` is a *pointer*, as that may accidentally lead to memory leaks.

They would prefer a name like `Point3DPtr` or alternatively not including the `*` in the `typedef` and requiring the client to declare each `Point3D` as a pointer.

```
Point3D *p = create_Point3D(2,3,5);
```

The Linux programming style guide recommends avoiding `typedefs` altogether.

The full structure definition is required in the module *implementation*.

```
// point3d.c
#include "point3d.h"

struct point3d {    // we used an array,
  int coord[3];     // but we could change our mind (flexibility)
};

Point3D create_Point3D(int x, int y, int z) {
  Point3D new = malloc(sizeof(struct point3d));
  new->coord[0] = x; new->coord[1] = y; new->coord[2] = z;
  return new;
}

int get_x(Point3D p) {
  return p->coord[0];
}

void set_x(Point3D p, int x) {
  p->coord[0] = x;
}
```

We presented the `Point3D` to illustrate how ADTs can be implemented in C, but `Point3D` is not a "typical" ADT as it stores a fixed quantity of data.

An ADT designed to store an **arbitrary** number of items is known as a ***collection ADT***. *Collection ADTs* are the most common ADTs.

All of the remaining ADTs discussed in this unit are collection ADTs.

In some contexts, an ADT is implicitly a collection ADT, and by some definitions, they are the *only* type of ADT.

An ADT is the combination of data and functions that "operate" on that data.

In *object-oriented programming* (CS 246), a ***class*** can encapsulate this combination of data and functions (methods).

Classes have additional features (behaviours) that extend beyond ADTs.

Stack ADT

The stack ADT is a collection of items that are "stacked" on top of each other. Items are *pushed* onto the stack and *popped* off of the stack. A stack is known as a LIFO (last in, first out) system. Only the most recently pushed item is accessible.

The stack ADT is not the same as the (call) stack section of memory, but conceptually they are very similar. Each item in the call stack is a frame. Each function call *pushes* a new frame on to the call stack, and each `return` *pops* the frame off of the call stack. The last function called is the first returned.

Stacks are often used in browser histories (“back”) and text editor histories (“undo”). In some circumstances, a stack can be used to “simulate” recursion.

Typical stack ADT operations:

- **push(s,i)** (or *add_front*) adds an item to the stack
- **pop(s)** (or *remove_front*) removes the “top” item on the stack
- **top(s)** (or *peek*) returns the next item to be popped
- **is_empty(s)** checks if the stack is empty

An ADT interface should be well documented and include run-times and examples of how to use the ADT (we have been terse here).

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

// create_Stack()    returns a new Stack
// destroy_Stack(s)  frees all of the memory
Stack create_Stack(void);
void destroy_Stack(Stack s);

// push(s, i) puts i on top of the stack
void push(Stack s, int i);

// pop(s) removes the top and returns it
int pop(Stack s);

// top(s) returns the top but does not pop it
int top(Stack s);

bool is_empty(Stack s);
```

```c
// stack.c
#include "stack.h"
#include "llist.h"     // our stack is a linked list

// The opaque structure is a natural ``wrapper''.
struct stack {
  struct llnode *topnode;
};

Stack create_Stack(void) {
  Stack new = malloc(sizeof(struct stack));
  new->topnode = NULL;
  return new;
}

bool is_empty(Stack s) {
  return s->topnode == NULL;
}

void push(Stack s, int i) {
  s->topnode = cons(i, s->topnode);
}
```

```
int top(Stack s) {
  assert(!is_empty(s));
  return s->topnode->item;
}

int pop(Stack s) {
  assert(!is_empty(s));
  int ret = s->topnode->item;
  struct llnode *backup = s->topnode;
  s->topnode = s->topnode->next;
  free(backup);
  return ret;
}

void destroy_Stack(Stack s) {
  while (!is_empty(s)) {
    pop(s);
  }
  free(s);
}
```

Beyond integers

The previous implementation only supported integers.

What if we want to have a stack of a different type?

There are three common strategies to solve this problem:

- create a separate implementation for each possible item type,
- use a `typedef` to define the item type, or
- use a `void` pointer type (`void *`).

The first option becomes very unwieldy. We will first discuss the `typedef` strategy, and then discuss the `void *` strategy later in this unit.

We don't have this problem in Racket because of dynamic typing.

This is one reason why Racket and other dynamic typing languages are so popular.

Some static typing languages have a *template* feature to avoid this problem. For example, in C++ a stack of integers is declared as:

```
stack<int> my_int_stack ;
```

The stack ADT (called a stack "container") is actually built-in to the C++ STL (standard template library).

The "typedef" strategy is to define the type of each item (ItemType) in a separate header file ("item.h") that can be provided by the client.

```
// item.h
typedef int ItemType;              // for stacks of integers
```

or...

```
// item.h
typedef struct posn ItemType;   // for stacks of posns
```

The ADT module would then be implemented with this ItemType.

```
#include "item.h"
void push(Stack S, ItemType i) {...}
ItemType top(Stack s) {...}
```

Having a client-defined `ItemType` is a popular approach for small applications, but it does not support having two different stack types in the same application.

The `typedef` approach can also be problematic if `ItemType` is a pointer type and it is used with dynamic memory. In this case, calling `destroy_Stack` may create a memory leak.

Memory management issues are even more of a concern with the third approach (`void *`), which is very similar to:

```
// item.h
typedef void *ItemType;   // each item is a void *
```

We will discuss this approach after introducing more ADTs.

Queue ADT

A queue is like a “lineup”, where new items go to the “back” of the line, and the items are removed from the “front” of the line. While a stack is LIFO, a queue is FIFO (first in, first out).

Typical queue ADT operations:

- **add_back(q,i)** (or *push_back, enqueue*)
- **remove_front(q)** (or *pop_front, dequeue*)
- **front(q)** (or *peek, next*) returns the item at the front
- **is_empty(q)**

Sequence ADT

The sequence ADT is useful when you want to be able to retrieve, insert or delete an item at any position in the sequence.

Typical sequence ADT operations:

- **item_at(s,k)** returns the item at position k (k $<$ length)
- **insert_at(s,k,i)** inserts i at position k and increments the position of all items after k
- **remove_at(s,k)**
- **length(s)** (or *size*)

Dictionary ADT

The dictionary ADT (also called a *map, associative array, or symbol table*), is a collection of **pairs** of **keys** and **values**. Each *key* is unique and has a corresponding value, but more than one key may have the same value. Dictionaries are unsequenced, and the primary design consideration is usually to achieve fast *lookups*.

Typical dictionary ADT operations:

- **lookup(d,k)** returns the value for a given key k, or "not found"
- **insert(d,k,v)** (or *add*) inserts a new key value pair (or replaces)
- **remove(d,k)** (or *delete*) removes a key and its value

Set ADT

The set ADT is similar to a mathematical set (or a dictionary with no values) where every item is unique. Sets are unsequenced and usually *sorted* (the unsorted ADT is less common).

Typical set ADT operations include: **member(s,i)** (*is_element_of*), **add(s,i)**, **union(s1,s2)**, **intersection(s1,s2)**, **difference(s1,s2)**, **is_subset(s1,s2)**, **size(s)**.

Set ADTs (and other ADTs) often have an **extract** (or *enumerate*) operation which will produce an array (or list) of all of the items.

Because many set ADT operations produce new sets and *copies* of items, they are more common in languages with garbage collection.

Implementing collection ADTs

A significant benefit of a collection ADT is that a client can use it "abstractly" without worrying about how it is are implemented.

ADT modules are usually well-written, optimized and have a well documented interface.

However, in this course, we are interested in how to implement ADTs.

"Typical" ADT implementations:

- **Stack**: linked lists or dynamic arrays
- **Queue**: linked lists
- **Sequence**: linked lists or dynamic arrays. Some libraries provide two different ADTs (*e.g.* a list ADT and a vector ADT) that provide the same interface but have different operation run-times.
- **Dictionary** and **Set**: self-balanced BSTs, or a hash table (a combination of arrays and linked lists – more on this in CS 240).

Selecting an ADT

Similar to selecting a data structure, choosing the right ADT is an important part of program **design**. Choosing the right ADT can make your code significantly more straightforward and efficient. Choosing the wrong ADT can be frustrating and inefficient.

It is also important to recognize when to use a simple data structure, and when to create or modify an existing data structure (or ADT) to better suit your specific needs.

Furthermore, you may want to combine ADTs (or combine ADTs and data structures). Your application may benefit from a stack of trees, or an array of queues.

void pointers

The `void` pointer (`void *`) is the closest C has to a "generic" type, which makes it suitable for ADT implementations.

`void` pointers can point to "any" type, and are essentially just memory addresses. They can be converted to any other type of pointer, but they can not be directly dereferenced.

```
int i = 42;
void *vp = &i;
int j = *vp;      // INVALID
int *ip = vp;
int k = *ip;      // VALID
```

While some C conversions are *implicit* (*e.g.*: `char` to `int`), there is a C language feature known as ***casting***, which *explicitly* "forces" a type conversion.

To cast an expression, place the destination type in parentheses to the left of the expression. This example casts a "`void *`" to an "`int *`", which can then be dereferenced

```
int i = 42;
void *vp = &i;
int j = *(int *)vp;
```

A useful application of casting is to avoid integer division when working with floats (see CP:AMA 7.4).

```
float one_half = ((float) 1) / 2;
```

There are two complications that arise from implementing ADTs with `void` pointers: memory management, and item comparisons.

An ADT module must establish and properly communicate the "rules" (or protocol) for memory management: **who is responsible for freeing the data?** The ADT or the client?

When the ADT is responsible for `free`ing, operations such as `pop` do not return the item (it would be invalid as it would have already been `free`d). It is the client's responsibility to retrieve the item through `top` and make a copy (backup) if it is needed after the `pop`.

One problem that occurs when the ADT is responsible is that the item itself may be a structure that contains additional pointers to dynamic memory (consider a stack of trees).

To avoid this problem, the ADT `create` function can have a parameter that is a pointer to a function that will be called to `free` each item. For a simple item, the `free` function can be passed to `create`, otherwise a more complex function (such as `free_tree`) can be passed.

A flexible ADT may accept a `NULL` pointer, which indicates that the client will be responsible for `free`ing the items.

When the client is responsible, operations such as `pop` and `remove_at` usually `return` a pointer to the item being removed so the client can `free` it.

This is slightly more complicated with a dictionary, where `delete` would `return` the key, and the client would have to first call `lookup` to `free` the value. Alternatively, `delete` could `return` a `struct` with two `void` pointers or have parameters that are pointers to pointers.

In addition, all ADT `destroy` functions would require that the ADT is empty, otherwise it will cause a memory leak.

The other problem with `void` pointers occurs when the ADT must make comparisons between items (*e.g.*: `search`).

Similar to the `free` function mentioned earlier, the `create` function would have a parameter that is a pointer to a function that will be called for comparisons.

In C, comparison functions follow the `strcmp(a,b)` convention, where `return` values of -1, 0 and 1 correspond to (a < b), (a == b), and (a > b) respectively.

A similar problem arises in the `typedef` approach, where the user would have to define a `compare_ItemType` function.

C generic algorithms

C provides a `qsort` function (part of <`stdlib.h`>) that can sort an array of any type. This is known as a "generic" algorithm.

`qsort` has a function (pointer) parameter that is used identically to the approach previously described for comparisons in ADTs.

```
void qsort(void *arr, int len, size_t size,
           int (*compare)(void *, void *));
```

The other parameters of `qsort` are an array of any type, the length of the array (number of elements), and the `sizeof` each element.

`qsort` example:

```
int compare_ints (const void *a, const void *b) {
  const int *ia = a;
  const int *ib = b;
  if (*ia < *ib) { return -1; }
  if (*ia > *ib) { return 1; }
  return 0;
}

int main(void) {
  int a[7] = {8, 6, 7, 5, 3, 0, 9};
  qsort(a, 7, sizeof(int), &compare_ints);
  //...
}
```

There is also a `bsearch` function that will search any sorted array for a key, and either return a pointer to the element if found, or `NULL` if not found.

```
void *bsearch(void *key, void *arr, int len, size_t size,
              int (* compare)(void *, void *));
```

Goals of this module

You should be familiar with the ADT terminology introduced.

You should be familiar with the collection ADTs introduced (stack, queue, sequence, dictionary, set) and their typical operations.

You should be able to implement any of the collection ADTs or be able to use them as a client.

You should be able to determine an appropriate data structure or ADT for a given design problem.

You should be able to reason about the running time of an ADT operation for a particular data structure implementation.

You should understand opaque structures and `typedef`.

You should understand the memory management issues related to using `void` pointers in ADTs and how `void` pointer comparison functions can be used with generic ADTs and generic algorithms.