Abstract Data Types (ADTs)

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

Selecting a data structure

In Computer Science, every data structure is some combination of the following "**core**" data structures.

- primitives (*e.g.,* an `int`)
- structures (*i.e.,* `struct`)
- arrays
- linked lists
- trees
- graphs

Selecting an appropriate data structure is important in **program design**. Consider a situation where you are choosing between an array, a linked list, and a BST. Some design considerations are:

- How frequently will you add items? remove items?
- How frequently will you search for items?
- Do you need to access an item at a specific position?
- Do you need to preserve the "original sequence" of the data, or can it be re-arranged?
- Can you have duplicate items?

Knowing the answers to these questions and the efficiency of each data structure function will help you make design decisions.

Sequenced data

Consider the following strings to be stored in a data structure.

```
"Bob" "Alice" "Charlie"
```

Is the **original sequencing** important?

- If it's the result of a competition, yes: `"Bob"` is in first place.
  We call this type of data ***sequenced***.
- If it's a list of friends to invite to a party, it is not important.
  We call this type of data ***unsequenced*** or "rearrangeable".

If the data is sequenced, then a data structure that *sorts* the data (*e.g.,* a BST) is likely not an appropriate choice. Arrays and linked lists are better suited for sequenced data.

Data structure comparison: sequenced data

Function	Dynamic 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`	$O(1)^*$	$O(n)^\dagger$
`remove_at`	$O(n)$	$O(n)$
`remove_front`	$O(n)$	$O(1)$
`remove_back`	$O(1)$	$O(n)^\diamond$

* amortized

† $O(1)$ with a wrapper strategy and a `back` pointer

$^\diamond$ $O(1)$ with a *doubly* linked list.

Data structure comparison: unsequenced (sorted) data

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

* $O(h)$ with a `size` augmentation.

† $O(\log n)$ with a `size` augmentation.

`select(k)` finds the k-th smallest item in the data structure.

For example, `select(0)` finds the smallest element.

example: design decisions

- An array is a good choice if you frequently access elements at specific positions (random access).
- A linked list is a good choice for sequenced data if you frequently add and remove elements at the start.
- A self-balancing BST is a good choice for unsequenced data if you frequently search for, add and remove items.
- A sorted array is a good choice if you rarely add/remove elements, but frequently search for elements and select the data in sorted order.

Abstract Data Types (ADTs)

Formally, an ***Abstract Data Type (ADT)*** is a mathematical model for storing and accessing data through ***operations***. As mathematical models they transcend any specific computer language or implementation.

As discussed in Section 02, ADTs are **implemented** as data storage modules that only allows access to the data through the interface functions (ADT *"operations"*).

The underlying data structure and implementation of an ADT is hidden from the client.

example: account ADT

To re-create the “account” ADT example from Section 02 in C we define an account structure.

```
struct account {
  char *username;
  char *password;
};
```

`create_account` returns a *pointer* to a **new** account. In addition, it makes **duplicates** of the username and password strings provided by the client.

```
struct account *create_account(const char *username,
                               const char *password) {

  struct account *a = malloc(sizeof(struct account));

  a->username = malloc((strlen(username)+1)*sizeof(char));
  strcpy(a->username, username);

  a->password = malloc((strlen(password)+1)*sizeof(char));
  strcpy(a->password, password);

  return a;
}
```

In C, our ADT also requires a `destroy_account` to `free` the memory created.

```
void destroy_account(struct account *a) {
  free(a->username);
  free(a->password);
  free(a);
}
```

The remaining operation `is_correct_password` verifies passwords.

```
bool is_correct_password(struct account *a,
                         const char *password) {
  return (strcmp(a->password,password) == 0);
}
```

The interface code for our ADT would be:

```
struct account {
  char *username;
  char *password;
};

struct account *create_account(const char *username,
                               const char *password);

void destroy_account(struct account *a);

bool is_correct_password(struct account *a,
                         const char *password);
```

In this interface, the structure is “transparent” and visible to the client. The client can access the `username` and `password` fields directly.

This is not good information hiding.

Opaque structures in C

C supports **opaque structures** through ***incomplete declarations***, where a structure is *declared* without any fields. With incomplete declarations, only *pointers* to the structure can be defined.

```
struct account;                 // INCOMPLETE DECLARATION

struct account my_account;      // INVALID

struct account *a;              // VALID
```

By providing only an incomplete declaration in the interface we achieve information hiding and gain security and flexibility.

To further improve our interface, we can create a new `Account` type that is a pointer to a `struct account`.

typedef

The C `typedef` keyword allows you to create your own "type" from previously existing types. This is typically 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" with `typedef`.

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

Account: new interface

```
struct account;

typedef struct account *Account;

// operations:

Account create_account(const char *username,
                       const char *password);

void destroy_account(Account a);

bool is_correct_password(Account a, const char *password);
```

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

A compromise is to use a type name that reflects that the type is a pointer (*e.g.*, `AccountPtr`).

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

Collection ADTs

We presented the `Account` ADT to illustrate how an ADT can be implemented in C, but as we discussed in Section 02, `Account` is not a “typical” ADT.

In this section we present five ***Collection ADTs*** that are designed to store an arbitrary number of items: stack, queue, sequence, dictionary and set.

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

stack: interface

```
// A real ADT would be MUCH better documented
struct stack;
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);
```

stack: implementation

```
#include "stack.h"

struct stack {
  struct llnode *topnode;     // our stack is a linked list
};

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

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

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

```c
void push(Stack s, int i) {
  struct llnode *new = malloc(sizeof(struct llnode));
  new->item = i;
  new->next = s->topnode;
  s->topnode = new;
}

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);
}
```

In this imperative example, the `pop` operation returned the value that was being "popped".

In a functional approach, any operation to "change" the stack (*i.e.,* `push` and `pop`) would return the **new** stack, and so `pop` would return the new stack instead of the value being 'popped".

In a functional approach, the `top` operation is more important (and essential before a `pop`).

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 are often used when fast *lookups* are required.

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 (element) 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)**.

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.

Typically, the collection ADTs are implemented as follows.

- **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 and a vector) that provide the same interface but have different operation run-times.
- **Dictionary** and **Set**: self-balanced BSTs or hash tables*.

* A hash table is typically a an array of linked lists (more on hash tables in CS 240).

Beyond integers

The stack implementation we presented only supported integers.

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

There are three common strategies to solve this problem in C:

- 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 is unwieldy and unsustainable. We first discuss the `typedef` strategy, and then the `void *` strategy.

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 statically typed languages have a *template* feature to avoid this problem. For example, in C++ a stack of integers is defined as:

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

The stack ADT (called a stack "container") is 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 ints
```

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 *`).

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 cannot 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;
```

Implementing ADTs with void pointers

There are two complications that arise from implementing ADTs with `void` pointers:

- **Memory management** is a problem because a protocol must be established to determine if the client or the ADT is responsible for freeing item data.
- **Comparisons** are a problem because some ADTs must be able to compare items when searching and sorting.

Both problems also arise in the `typedef` approach.

The solution to the **memory management** problem is to make the *ADT interface explicitly clear* whose responsibility it is to `free` any item data: the client or the ADT. Both choices present problems.

For example, when it is the **client's responsibility** to `free` items, care must be taken to retrieve and `free` every item before a `destroy` operation, otherwise `destroy` could cause memory leaks. A precondition to the `destroy` operation could be that the ADT is empty (all items have been removed).

When it is the **ADT's responsibility**, problems arise if the items contain additional dynamic memory.

For example, consider if we desire a sequence of accounts, where each account is an instance of the account ADT we implemented earlier. If the sequence `remove_at` operation simply calls `free` on the item, it creates a memory leak as the username and password are not freed.

To solve this problem, the client can provide a customized `free` function for the ADT to call (*e.g.,* `destroy_account`).

example: stack interface with void pointers

```
// (partial interface) CLIENT'S RESPONSIBILITY TO FREE ITEMS

// push(s, i) puts item i on top of the stack
//   NOTE: The caller should not free the item until it is popped
void push(Stack s, void *i);

// top(s) returns the top but does not pop it
//   NOTE: The caller should not free the item until it is popped
void *top(Stack s);

// pop(s) removes the top item and returns it
//   NOTE: The caller is responsible for freeing the item
void *pop(Stack s);

// destroy_Stack(s) destroys the stack
// requires: The stack must be empty (all items popped)
void destroy_Stack(Stack s);
```

example: client interface

```
#include "stack.h"

// this program reverses the characters typed
int main(void) {
  Stack s = create_Stack();
  while(1) {
    char c;
    if (scanf("%c", &c) != 1) break;
    char *newc = malloc(sizeof(char));
    *newc = c;
    push(s,newc);
  }
  while(!is_empty(s)) {
    char *oldc = pop(s);
    printf("%c", *oldc);
    free(oldc);
  }
  destroy_Stack(s);
}
```

Comparison functions

The dictionary and set ADTs often *sort* and *compare* their items, which is a problem if the item types are `void` pointers.

To solve this problem, we can provide the ADT with a ***comparison function*** (pointer) when the ADT is created.

The ADT would then just call the comparison function whenever a comparison is necessary.

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 comparison function for integers
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;
}
```

A `typedef` can be used to make declarations less complicated.

```
typedef int (*CompFuncPtr) (const void *, const void *);
```

example: dictionary

```
// dictionary.h (partial interface)

struct dictionary;
typedef struct dictionary *Dictionary;

typedef int (*DictKeyCompare) (const void *, const void *);

// create a dictionary that uses key comparison function f
Dictionary create_Dictionary(DictKeyCompare f);

// lookup key k in Dictionary d
void *lookup(Dictionary d, void *k);
```

```c
// dictionary.c (partial implementation)

struct bstnode {
  void *key;
  void *value;
  struct bstnode *left;
  struct bstnode *right;
};

struct dictionary {
  struct bstnode *root;
  DictKeyCompare key_compare;   // function pointer
};


Dictionary create_Dictionary(DictKeyCompare f) {
  struct dictionary *newdict = malloc(sizeof(struct dictionary));
  newdict->key_compare = f;
  //...
}
```

This implementation of `lookup` illustrates how the comparison function would work.

```
void *lookup(Dictionary d, void *k) {
  struct bstnode *t = d->root;
  while (t) {
    int result = d->key_compare(k, t->key);
    if (result < 0) {
      t = t->left;
    } else if (result > 0) {
      t = t->right;
    } else { // key found!
      return t->value;
    }
  }
  return NULL; // (no key found)
}
```

C generic algorithms

Now that we are comfortable with `void` pointers, we can use C's built-in `qsort` function.

`qsort` is part of <`stdlib.h`> and can sort an array of any type.

This is known as a "generic" algorithm.

`qsort` requires a comparison function (pointer) that is used identically to the comparison approach we described for 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.

example: qsort

```
// see previous definition
int compare_ints (const void *a, const void *b);

int main(void) {

  int a[7] = {8, 6, 7, 5, 3, 0, 9};

  qsort(a, 7, sizeof(int), compare_ints);

  //...
}
```

C also provides a generic binary search (`bsearch`) function that searches 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 Section

At the end of this section, you should be able to:

- describe the collection ADTs introduced (stack, queue, sequence, dictionary, set) and their operations
- implement any of the collection ADTs or be able to use them as a client
- determine an appropriate data structure or ADT for a given design problem
- deduce the running time of an ADT operation for a particular data structure implementation

- use opaque structures (incomplete declarations) and `typedef`
- describe 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