Abstract Data Types (ADTs)

Design & Generic Implementations

Optional Textbook Readings: CP:AMA 19.5, 17.7 (`qsort`)

The primary goals of this section are to be able to use choose appropriate data structures and/or ADTs for a given situation, and to write generic ADTs.

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.

```
"Wei" "Jenny" "Ali"
```

Is the **original sequencing** important?

- If it's the result of a competition, yes: `"Wei"` 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`	$O(1)^*$	$O(1)^\dagger$
`remove_at`	$O(n)$	$O(n)$
`remove_front`	$O(n)$	$O(1)$
`remove_back`	$O(1)$	$O(1)^\diamond$

$*$ amortized

$\dagger$ requires a `back` pointer – $O(n)$ without

$\diamond$ requires a *doubly* linked list and a `back` pointer – $O(n)$ without.

Data structure comparison: unsequenced (sorted) data

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

† requires a `count` augmentation – $O(n)$ without.

`select(k)` finds the item with index k in the 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.

Implementing collection ADTs

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

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

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**s): self-balanced BSTs or hash tables*.

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

Beyond integers

In Section 10, we implemented a Stack ADT that stores `int` items.

What if we want a stack that stores a different *type* of item?

We could write a separate implementation for each possible item type, but that is unwieldy.

It would be nice if we could develop a “generic” ADT that can store “any” type.

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).

void pointers

The `void` pointer (`void *`) is the closest C has to a “generic” type.

`void` pointers can store the address of (“point at”) **any** type of data.

```
int i = 42;
int *pi = &i;
struct posn p = {3, 4};

void *vp = NULL;
vp = &i;
vp = &p;
vp = pi;
vp = &pi;
vp = &vp;
```

> `void` pointers can not point at functions.

Dereferencing void pointers

`void` pointers **can not be dereferenced**.

```
int i = 42;
void *vp = &i;

int j = *vp;        // INVALID
*vp = 13;           // INVALID
int k = vp[0];      // INVALID
```

Do not confuse `void` *functions* (that return “nothing”) and `void` *pointers* (that can point at “anything”).

They are unrelated concepts that share a keyword.

Assigning from void pointers

`void` pointers can not be dereferenced.

However, the address stored in a `void` pointer can be assigned to the **correct type** of pointer variable and *then* be dereferenced.

This is why `malloc` works for any type of data (its return type is a `void` pointer).

```
int i = 42;

void *vp = &i;              // vp points at an int

int *ip = vp;               // OK

int j = *ip;
*ip = 13;
```

Caution with void pointers

When assigning from `void` pointers, we lose *type safety* and could have a pointer pointing to the wrong type. Dereferencing such a pointer causes undefined and potentially dangerous behaviour.

```
int i = 42;
void *vp = &i;              // vp points at an int

struct posn *pp = vp;       // BAD (but allowed by C)

int j = pp->y;              // BAD (undefined behaviour)
pp->y = 13;                 // BAD (undefined behaviour)
```

You should only assign from a `void` pointer **when you are sure** that the pointer variable is the **correct type** for the data at that address.

If `void` pointers are potentially unsafe, why are they used?

Generic functions

With `void` pointers, we can now write functions that operate on *any* type of data.

For example, the `qsort` function is part of <`stdlib.h`> and can sort an array of **any type** (in ascending order).

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

The parameters are:

- an `arr`ay, its `len`gth and the `size` of each element (in bytes)
- a function to `compare` elements

Generic comparison function

The comparison function required by `qsort` follows the same convention as `strcmp`.

In other words, `compare(a, b)` returns:

- a negative `int` if `*a` *precedes* `*b`
- zero if `*a` and `*b` are equal.
- a positive `int` if `*a` *follows* `*b`

If you reverse the above behaviour, then `qsort` will sort an array in *descending* order.

example: qsort with ints

```
// requires: a, b point at ints
int compare_ints(const void *a, const void *b) {
  const int *ia = a;
  const int *ib = b;
  return *ia - *ib;
}

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

```
8, 6, 7, 5, 3, 0, 9.
0, 3, 5, 6, 7, 8, 9.
```

example: qsort with chars

```c
// requires: a, b point at chars
int compare_chars(const void *a, const void *b) {
  const char *ia = a;
  const char *ib = b;
  return *ia - *ib;
}

int main(void) {
  char s[] = "sortable";
  printf("%s\n", s);
  qsort(s, strlen(s), sizeof(char), compare_chars);
  printf("%s\n", s);
}
```

```
sortable
abelorst
```

More generic functions

<stdlib.h> also provides a generic binary search (bsearch) that either returns a pointer to the key in the **sorted** array if found, or NULL if not found.

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

Another useful function is memcpy in <string.h> which simply copies size bytes from a source to a destination.

```
void *memcpy(void *dest, const void *src, size_t size);
```

memcpy returns the destination for historical reasons.

Generic map function

In Section 07, we wrote an `int` map function but now we can write a *generic* version.

The only challenge is finding the address of each element (`arr[i]`).

Because a `char` is only one byte, we can "pretend" the array is a `char` array, and then perform the pointer arithmetic **manually**.

```
void generic_map(void *arr, int len, size_t size,
                 void (*map_function)(void *)) {
  char *arr_base = arr;
  for (int i = 0; i < len; ++i) {
    map_function(arr_base + i * size);
  }
}
```

example: generic map

```c
// requires: i points at an int
// effects: modifies *i
void apply_sqr(void *i) {
  int *pi = i;
  *pi = *pi * *pi;
}

int main(void) {
  int a[7] = {8, 6, 7, 5, 3, 0, 9};
  print_array(a, 7);
  generic_map(a, 7, sizeof(int), apply_sqr);
  print_array(a, 7);
}
```

```
8, 6, 7, 5, 3, 0, 9.
64, 36, 49, 25, 9, 0, 81.
```

Generic ADTs with void pointers

We can now create *generic* container ADTs that can store *“any”* type of data by storing `void` pointers.

For example, we can modify our `stack` ADT from Section 10 to store `void` pointer items (instead of `int`s).

This requires only a few minor changes to the code.

As we will discuss shortly, the only significant change is the behaviour of `stack_destroy`, which now requires the `stack` to be **empty**.

example: generic stack

Very little code has changed (indicated by `//!`).

```
struct stack;

struct stack *stack_create(void);

bool stack_is_empty(const struct stack *s);

void stack_push(void *item, struct stack *s);               //!

const void *stack_top(const struct stack *s);               //!

void *stack_pop(struct stack *s);                           //!

// requires: s is empty                                     //!
void stack_destroy(struct stack *s);
```

Previously, we used an array of `int`s. Now, we want an array of `void` pointers:

```
struct int_stack {
  int len;
  int maxlen;
  int *data;           // OLD
};

struct stack {
  int len;
  int maxlen;
  void **data;         //! (NEW)
};
```

A “`void *`” pointer is generic and can point at “anything”.

A “`void **`” pointer is *not* generic: it is a pointer to a “`void *`”, and can be dereferenced and behave like an array (of `void` pointers).

```c
struct stack *stack_create(void) {
  struct stack *s = malloc(sizeof(struct stack));
  s->len = 0;
  s->maxlen = 1;
  s->data = malloc(s->maxlen * sizeof(void *));              //!
  return s;
}


// requires: s is empty                                      //!
void stack_destroy(struct stack *s) {
  assert(stack_is_empty(s));                                 //!
  free(s->data);
  free(s);
}


bool stack_is_empty(const struct stack *s) {
  return s->len == 0;
}
```

```
void stack_push(void *item, struct stack *s) {                  //!
  if (s->len == s->maxlen) {
    s->maxlen *= 2;
    s->data = realloc(s->data, s->maxlen * sizeof(void *));  //!
  }
  s->data[s->len] = item;
  s->len += 1;
}


const void *stack_top(const struct stack *s) {                  //!
  assert(!stack_is_empty(s));
  return s->data[s->len - 1];
}


void *stack_pop(struct stack *s) {                              //!
  assert(!stack_is_empty(s));
  s->len -= 1;
  return s->data[s->len];
}
```

example: client for a generic stack

```c
// this reads in strings ("words") and then prints them
// out in reverse order.

int main(void) {
  struct stack *s = stack_create();
  while(1) {
    char *str = read_str();  // from Sec 10
    if (!str) {
      break;
    }
    stack_push(str, s);
  }
  while(!stack_is_empty(s)) {
    char *str = stack_pop(s);
    printf("%s\n", str);
    free(str);
  }
  stack_destroy(s);
}
```

Memory management with generic ADTs

A generic ADT does not "know" the type of the items it stores, and so it does not "know" the *type of memory* those items are using.

For example, each item could be a pointer to:

- non-heap memory (*e.g.,* a string literal)
- a heap allocation (*e.g.,* a dynamically allocated array)
- a structure with multiple dynamic elements (*e.g.,* a linked list)

Consequently, the ADT can not perform any *memory management* on its items (*e.g.,* the ADT can not naïvely `free` an item).

For our generic stack ADT, we avoided the memory management issue by *requiring* the stack to be *empty* before it is destroyed.

This means the client is responsible for `free`ing all of the items.

If we did not add this requirement, destroying the stack would cause a **memory leak** if the items use dynamic memory.

Alternatively, we could design the stack ADT so the client has to provide a `free` function (that appropriately `free`s an item).

example: generic stack ADT with free function

```
// store pointer to the free function in the ADT structure

struct stack {
  int len;
  int maxlen;
  void **data;
  void (*free_item)(void *);    // function pointer
};

// client provides free function when stack is created

struct stack *stack_create(void (*free_function)(void *)) {
  struct stack *s = malloc(sizeof(struct stack));
  s->len = 0;
  s->maxlen = 1;
  s->data = malloc(s->maxlen * sizeof(void *));
  s->free_item = free_function;
  return s;
}
```

```c
// note: no longer requires stack to be empty
void stack_destroy(struct stack *s) {
  for (int i = 0; i < s->len; ++i) {       // free all of
    s->free_item(s->data[i]);              // the items
  }
  free(s->data);
  free(s);
}
```

The client can provide an appropriate `free` function.

```c
// for non-heap memory (e.g., string literals)
void free_nothing(void *item) { } // do nothing

// for dynamic arrays
void free_single_allocation(void *item) {
  free(item);
}

// for data types with multiple dynamic elements
void free_linked_list(void *item) {
  list_destroy(item);
}
```

Client communication

For a simple stack ADT, a `free` function is unwarranted. Requiring an empty stack before destruction is a reasonable approach.

For other ADTs, it may be necessary. For example, a dictionary ADT might require *two* `free` functions (for keys *and* values).

Other ADTs may have operations that copy (or “duplicate”) items, and require a `copy` function.

> It is important to make a **clear interface** to communicate to the client how the memory for the individual items will be managed.

Comparison functions

Many ADTs need a **comparison function** to compare items and possibly *sort* items.

For example, a dictionary ADT must to be able to compare *keys* to perform a `lookup` operation.

Typically, comparison functions use the same convention as `strcmp` (and `qsort`).

As with the earlier example (providing a `free` function to the stack ADT), a comparison function pointer can be provided when the ADT is created and stored in the ADT structure.

example: dictionary ADT with comparison function

```
struct dictionary {
  // ...
  int (*key_compare)(const void *, const void *);   // func ptr
};

const void *dict_lookup(const void *key,
                        const struct dictionary *d) {
  const struct bstnode *node = d->root;
  while (node) {
    int result = d->key_compare(key, node->item);
    if (result == 0) return node->value;
    if (result < 0) {
      node = node->left;
    } else {
      node = node->right;
    }
  }
  return NULL;
}
```

Alternatives to C

We have reached our final core learning goal: making *generic* ADTs in C with `void` pointers. Much of the course content has been leading up to this point.

However, it seems appropriate to mention that C is not the most suitable language for working with generic ADTs. C is a very powerful language, but it is sometimes inconvenient and inflexible.

There are many other languages that are easier to work with and have better support for information hiding, abstraction, modularization, dynamic types, garbage collection, *etc.*.

It may surprise you to know that the Racket language interpreter is written in C (it is available on `github`).

How is it possible that Racket uses dynamic typing?

(and garbage collection and `int`s that do not overflow, and . . .)

Consider this (simplified) C representation of a Racket object:

```
struct racket_object {
  int type;              // int = 1, string = 2, list = 3, etc.
  void *data;
}
```

Each object has an `int` to identify the *type* and a `void` pointer to the data. Built-in Racket functions simply check the type and then process the data accordingly.

In practice, you may see C code that ***casts*** `void` pointers.

Casting explicitly “forces” a *type conversion* by placing the destination type in parentheses to the left of an expression.

C does not require `void` pointers to be cast but C++ does, which is why it is often seen in practice.

A useful application of casting is to avoid integer division when working with `float`s or `double`s (see CP:AMA 7.4).

Here are a few examples of casting:

```
double one_half = ((double) 1) / 2;
int *a = (int *) malloc(n * sizeof(int));
void *vp = (void *) a;
*(int *)vp = 136;
```

Goals of this Section

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

- determine an appropriate data structure and/or ADT for a given design problem
- implement a generic function with `void` pointers
- implement generic ADTs using comparison and memory management functions
- describe the memory management issues related to using `void` pointers in ADTs