Abstract Data Types (ADTs) & Design

Readings: CP:AMA 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 and a `back` pointer.

Data structure comparison: unsequenced (sorted) data

Function	Sorted Dynamic Array	Sorted Linked List	Regular BST	Self-Balancing BST
`select`	$O(1)$	$O(n)$	$O(n)^*$	$O(n)^\dagger$
`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.

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 presented an implementation of a Stack ADT that only supported a stack of `int`egers.

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

There are three common strategies to solve this “type” 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).

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).

```c
typedef int (* MapFn)(int);

int add1(int n) { return n+1; }

void array_map(MapFn f, int a[], int len) {  // <- cleaner!
  for (int i=0; i < len; ++i) {
    a[i] = f(a[i]);
  }
}

int main(void) {
  int arr[6] = {4, 8, 15, 16, 23, 42};
  array_map(add1, arr, 6);
  MapFn f = add1;
  array_map(f, arr, 6);
  //...
}
```

Stack ADT: cleaner interface

```
struct stack;

// use [Stack] instead of [struct stack *]
typedef struct stack *Stack;

// operations:

Stack stack_create(void);

bool stack_is_empty(Stack s);

int stack_top(Stack s);

int stack_pop(Stack s);

void stack_push(int item, Stack s);

void stack_destroy(Stack s);
```

Some programmers consider it poor style to use `typedef` to “abstract” that a type 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.*, `StackPtr`).

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

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 stack_push(Stack S, ItemType i);

ItemType stack_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

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

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

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

// stack_destroy(s) destroys the stack
// requires: The stack must be empty (all items popped)
void stack_destroy(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 dict_create(DictKeyCompare f);

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

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

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

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


Dictionary dict_create(DictKeyCompare f) {
  Dictionary *d = malloc(sizeof(struct dictionary));
  d->root = NULL;
  d->key_compare = f;
  return d;
}
```

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

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

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,
             CompFuncPtr f);
```

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,
              CompFuncPtr f);
```

Goals of this Section

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

- determine an appropriate data structure or ADT for a given design problem
- 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