Lecture 30: Data Structures, Queues & Graphs

10:00 AM, Nov 14, 2018

Contents

1 Data structures 1
  1.1 Analysis ................................................................. 1

2 Queues 2

3 Set/membership data structure 3

4 A short-circuiting-like search procedure 3

5 Graph search 4

Last lecture, I showed some video from a lecture by Barbara Liskov, a Turing Award Winner. She spoke at Brown some time ago. She was a pioneer in programming methodology and programming languages that support data abstraction.

Another Turing Award Winner, Robert Tarjan, will speak on December 6. He is a pioneer in data structures and algorithms.

1 Data structures

A data structure is a data object on which a series of operations are performed. Some operations ask questions, e.g.

- Is 7 in the set?
- What is the top element of the stack?
- What is x bound to in the environment?

These are called queries.

Other operations are called updates. When you are not doing functional programming (as in CS18), often an update will go in to the data object and change the data object, e.g.

- add/remove 7 from the set
- remove the top element of the stack,
- add or remove a binding of some variable to some value in an environment
Part of functional programming is that our program never actually changes any data object; instead, 
the program derives a new data object that differs slightly from the old, though often we informally 
use the same language to describe updates. For example, we say an operation pushes 7 onto a stack 
but in reality the push operation returns a slightly different stack in which 7 is the top.

1.1 Analysis

When we discussed algorithms, we said that $O(n)$ time (linear time) is good. For most algorithmic 
problems, one cannot expect sublinear-time algorithms. The reason is that, for most problems, 
knowing the output in the worst case depends on reading all of the input. Because the input size is 
n, at least $n$ operations are needed.

How do we analyze data structures? What would it mean to say that a data structure runs in $O(n)$ 
time? What does input size even mean in this context?

Often a data structure stores a bunch of items, in which case often we use $n$ to denote the number 
of items stored. Sometimes $n$ is used to denote the total number of operations performed on the 
data structure during its entire “life”.

Let’s consider the list-based implementation of STACK. Regardless of whether $n$ denotes the number 
of items stored or the total number of operations performed, each operation (query/update) takes 
$O(1)$ time in the worst case. That means the time required is bounded by a constant, independent 
of $n$. This is the best one can hope for in a data structure.

For some data structures, one cannot show such a good bound on the time per operation, but one can 
show a good bound on the average time per operation. For example, one might show that the 
total time for performing $n$ operations is at most $cn$ for some constant $c$. This means that the 
average number of operations is at most $\frac{cn}{n}$, which is $c$. This kind of analysis is called amortized 
analysis. We say that the worst-case amortized time per operation is $O(1)$. Robert Tarjan was one 
of the pioneers of amortized analysis.

2 Queues

You were asked to write a signature for queues, based on the signature for stacks. Here is my 
solution:

```ocaml
module type QUEUE = 
  sig 
    type 'a queue 
    val empty: 'a queue 
    val is_empty: 'a queue -> bool 
    val enqueue: 'a queue * 'a -> 'a queue 
    val peek: 'a queue -> 'a 
    val dequeue: 'a queue -> 'a queue 
  end
```

How to implement queues? There is a straightforward implementation using lists, which you will do 
in homework.
Queues are described as first-in, first-out (FIFO). Stacks are last-in, first-out (LIFO). LIFO works well with Ocaml lists but FIFO does not. In particular, the straightforward list-based solution takes $O(n)$ time per operation in the worst case. There is a data structure that takes $O(1)$ time per operation but it is not a functional data structure; it requires that data objects be mutated.

There is a functional approach that achieves amortized $O(1)$ time per operation. This means that in the average case of running many operations, each operation is constant time, even though some operations may take significantly more time. This involves including pointers in each element to the next and to the previous list, so that the beginning and end of the list can both be accessed quickly.

### 3 Set/membership data structure

Consider a data structure that can represent a set. Here is a signature:

```ocaml
module type SET =
  sig
    type 'a set
    val empty: 'a set
    val insert: 'a set * 'a -> 'a set
    val containsP: 'a set * 'a -> bool
  end
```

Again, there is an easy implementation using lists, but again the per-operation time is $O(n)$.

There is a solution using binary search trees for which the worst-case time per operation is $O(\log n)$.

There is a solution that even achieves $O(1)$ time in a sense.

You will learn about these in CS18.

### 4 A short-circuiting-like search procedure

Suppose $f$ is a procedure that, given a binding environment, returns true if the environment includes a binding for "password". Maybe the environment is represented by a list and is very big, so this could take a long time.

Suppose $f$ is a procedure that, given a binding environment, finds the value to which "password" is bound. Maybe the environment is represented by a list and is very big, so this could take a long time.

Now imagine you have a list consisting of ten thousand such environments, and you want to find the first environment in the list that contains "password", and return the corresponding value. You could maybe use map or fold but that could involve invoking $f$ on ten thousand environments, even if the first environment containing "password" occurs among the first few in the list. That seems wasteful.

Instead, you want $f$ to be invoked only until "password" is first found.
let rec successP = function
  __, [] -> false
  | f, hd :: tl -> f hd || successP (f, tl)

To take into account that the string might not be found, we use the variant type option:

type 'a option = Some of 'a | None

The procedure \( f \) should return an option.

More generally, suppose \( f \) is any time-consuming procedure that returns an option. We want a procedure \( \text{first_success} \) that takes \( f \) and a list \( \text{ins} \) of possible inputs to \( f \), and finds the first output that is not None. The procedure \( \text{first_success} \) should be “short-circuiting” in the sense that it does not apply \( f \) to every input in the list \( \text{ins} \).

Type of \( \text{first_success} \) is

\[
(\text{'a -> 'b option}) \times \text{'a list} \rightarrow \text{'b option}
\]

Example:

# search [("name", 1); ("robot", 5); ("password", 9)]
- : int option = Some 9

# first_success (search, [ ["hello", 10]; ("goodbye", 0)];
  ["name", 1]; ("robot", 5); ("password", 9)];
  ["move", 2]; ("flip", 3); ("now", 4)]
- : int option 9

let rec first_success = function
  __, [] -> None
  | f, a::rest ->
    match f a with
    Some b -> Some b
    | None -> first_success (f, rest) ;;

Quiz: Write \( \text{first_success} \) Solution:

let rec first_success = function
  __, [] -> None
  | f, a::rest ->
    match f a with
    Some b -> Some b
    | None -> first_success (f, rest) ;;

5 Graph search

Data structures are very useful because a single data structure can have many uses. We will look at graph search, and will discover that Set and Queue are useful in those things.
Searching in a graph can take different forms. One kind of graph search in a HUGE graph happens in your Game AI. Another kind of graph search happens, for example, when you are searching for a solution to a puzzle or finding a good route in a map or communication network.

First, let’s discuss graphs. A graph is a mathematical structure consisting of vertices (also called nodes) and edges (also called arcs—sometimes called links in less graph-theoretical settings). The singular of vertices is vertex.

In the interest of full disclosure, a big part of my research is focused on graph theory and the application of graph theory in algorithms that operate on graphs.

We draw vertices as small dots or circles, and we draw edges as lines between them, sometimes with arrowheads if we want to indicate a preferred or allowed direction. The origin of vertex/edge terminology (and arguably of graph theory) is polyhedra. I have a particular attachment to graphs that can arise in this way. However, graphs need not have a geometric interpretation. Applications of graphs are ubiquitous in computer science.

The neighbors of a vertex \( v \) are the vertices to which \( v \) is directly connected by an edge. The graph might not be symmetric—\( v \) might not be a neighbor of its neighbor.

We will represent a graph by procedure that, given a vertex, outputs the neighbors of the vertex. (That is not sufficient; we often also need a way to represent the set of all vertices of the graph, but I will elide this issue.)

One special kind of graph is a rooted tree; we have seen these before.

```
# let g = function n ->
  if n < 30 then [3*n; 3*n+1; 3*n+2] else [];;
val g : int -> int list = <fun>
```

input: graph g, start vertex , stop vertex  
output: is there a sequence \( v_1, \ldots, v_k \) of vertices with the following properties?

- \( v_1 \) is the start vertex,
• $v_k$ is the stop vertex, and
• for $i = 2, \ldots, k$, $v_i$ is a neighbor of $v_{i-1}$

Define a function `pathP` that finds whether there exists a path from one point to another on a graph.

```ocaml
define a function pathP that finds whether there exists a path from one point to another on a graph.

let rec pathP = function (g, origin, dest) ->
    origin = dest || successP ((function v -> pathP (g, v, dest)), (g origin))

Now we want a procedure to actually find the sequence $v_1, \ldots, v_n$. Of course, there could be no such sequence, so we should have the procedure return an option:

```ocaml
let rec find_path = function (g, origin, dest) ->
    if origin = dest then Some(dest)
    else match first_success ((function v -> find_path (c, v, dest)), g origin) with
        None -> None
    | Some path -> Some (origin::path)
```

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)