CS 2SC3 and SE 2S03 Fall 2008

05 Dynamic Data Structures

William M. Farmer

Department of Computing and Software
McMaster University

16 November 2008

What are Dynamic Data Structures?

- A static data structure is a data structure whose size is fixed during run time.
  - Overhead cost is low.
  - Expansion cost is high.
- A dynamic data structure is a data structure whose size changes during run time.
  - Overhead cost is high.
  - Expansion cost is low.

Program Memory

A machine code program has four kinds of memory:

1. Processor registers.
2. Static memory.
3. The stack (also called the call stack or execution stack).
4. The heap.

Persistence

- The persistence of an entity (e.g., a variable) is the period of time the entity is available to a running program.
- Examples:
  - The persistence of a running function procedure begins when it is called and ends when it returns a value.
  - The persistence of a variable declared in a procedure normally has the same persistence as the procedure.
  - The persistence of a global variable is from when it is first declared to the termination of the program.

Static Memory Allocation

- Static memory allocation is done at compile time.
  - Performed by the compiler.
  - The size of static memory does not change during run time.
- Static memory includes:
  - Program code.
  - Data that needs to be available for the lifetime of the program.
  - Global constants.
- There is no run time allocation overhead.
- Data structures held in static memory persist for the lifetime of the program.
- Static memory is not deallocated during run time.

Automatic Memory Allocation

- Automatic memory allocation is done at run time whenever a procedure is called.
  - A frame is pushed on the stack when the procedure is called.
  - The frame is popped from the stack when the procedure finishes.
- Stack memory includes:
  - Return address.
  - Local variables.
- Allocation overhead is modest.
- Data structures held in stack memory persist only for the lifetime of the procedure call.
- Stack memory is automatically deallocated when a procedure call finishes.

Dynamic Memory Allocation

- Dynamic memory allocation is done at run time when a data structure is created in the heap.
- The heap memory includes:
  - Dynamic data structures.
  - Data structures that need to persist longer than procedure calls.
- Allocation overhead is high.
- Data structures held in heap memory persist until the memory is deallocated.
- In OCaml, heap memory is implicitly deallocated by garbage collection.
- In C, heap memory is explicitly deallocated.

Heap Memory Allocation and Deallocation in C

- Heap memory is allocated using the operator `malloc`:

  ```
  type * ptr = malloc(sizeof(type));
  ```

- Heap memory is deallocated using the operator `free`:

  ```
  free(ptr);
  ```

- Problems:

  1. Unneeded heap memory is not freed by the programmer or the pointer to heap memory is lost (memory leak).
  2. Heap memory is accessed after it is freed.

Linked Lists

- A linked list is a dynamic data structure consisting of a sequence of linked nodes that holds a list of elements.
- Each node is a record of type t containing various data fields and one or two references of type t:
  1. A field named next that points to the "next" node.
  2. A field named previous that points to the "previous" node.

  Note: The type t is self-referential.
- A link list has four basic forms:
  1. Singly linked (having one of the next and previous fields).
  2. Doubly linked (having both of the next and previous fields).
  3. Singly linked in a circle.
  4. Doubly linked in a circle.

Linked List Operators

- Constructor: Starts the construction of a new linked list.
- Node selector: A node is accessed sequentially by following the links until the node is reached.
- Node mutators:
  - Node removal: A node is removed from the linked list.
  - Node insertion. A new node is inserted into the linked list.

Comparison of Linked Lists with Arrays

- Linked lists are more space efficient than arrays.
  - Arrays fill up; linked lists do not.
  - Arrays may have empty cells; linked lists do not.
  - Resizing an array is costly; resizing a linked list is not.
- Access to an array element is faster than to linked list elements.
  - Arrays support random access.
  - Linked lists support only sequential access.
- Linked lists have higher space and time overhead than arrays.
  - Cells requires more space to hold node references.
  - Removing and inserting elements takes more time.

Binary Trees

- A binary tree is a dynamic data structure consisting of a tree of linked nodes that holds a tree of elements.
- Each node is a record of type t containing various data fields and two references of type t:
  1. A field named left that points to the "left" child node.
  2. A field named right that points to the "right" child node.
- Data can be stored at each node or at only the leaf nodes.

Sum Types in OCaml

- A sum type represents a disjoint union of values.
  - Sum types are also called union types and variant types.
  - Enumerated types are a special case of sum types.
- Sum type declaration:

```
type name =
    Name_1 [of t_1]
  | Name_2 [of t_2]
      ⋮
  | Name_n [of t_n] ;;
```

- The Name_i are constructors that:
  1. Construct values of the sum type.
  2. Tag the values of the sum type to distinguish the components of the corresponding disjoint union.
  3. Select values of the sum type via pattern matching.

Summary of Types in OCaml

- Types of Immutable Values
  - Basic types: `unit`, `bool`, `int`, `float`, `char`.
  - Function types: t_1 `->` t_2.
  - Product types: t_1 `*` $\cdots$ `*` t_n.
  - List types: t `list`.
  - Sum types: Name_1 [`of` t_1] `|` $\ldots$ `|` Name_n [`of` t_n].
- Types of Mutable Values
  - Reference types: t `ref`.
  - Record types:
    $\{$[`mutable`] name_1 `:` t_1 `;` $\ldots$ `;` [`mutable`] name_n `:` $t_n\}$.
  - Array types: t `array`.
  - String type: `string`.

Type Declarations in OCaml

- (Recursive) type declarations:

  ```
  type name = type_expression ;;
  ```

- Mutually recursive type declarations:

  ```
  type name1 = type_expression1
  and  name2 = type_expression2
    ⋮
  and  namen = type_expressionn ;;
  ```

- Parameterized type declarations:

  ```
  type 'a name = type_expression ;;
  type ('a1, ..., 'an) name = type_expression ;;
  ```

Pattern Matching in OCaml (1/3)

- Pattern matching provides an easy way to access components of complex data structures.
- A pattern is an expression with zero or more free (unbound) variables.
- A pattern must be linear in the sense that each free variable cannot occur more than once in the pattern.
- Let p be a pattern containing the free variables $x_1, \ldots, x_n$.
- A value v matches p if there is a set of bindings for $x_1, \ldots, x_n$ such the value of p under these bindings equals v. Let us call this set of bindings the match bindings.
- The wildcard pattern _ matches every possible value.

Pattern Matching in OCaml (2/3)

- The following syntax is used to do pattern matching:

  ```
  match expr with
  | p1 -> expr1
    ⋮
  | pn -> exprn
  ```

  where the $p_1, \ldots, p_n$ are patterns of the same type as expr. Note: the first `|` is optional.

- A pattern matching expression is evaluated as follows:

  1. expr is evaluated and then sequentially compared with the patterns $p_1, \ldots, p_n$ until a match is found.
  2. If p_i is the first pattern that matches the value of expr, then the value of expr_i under the corresponding match bindings is returned.
  3. If no match is found, the `Match_failure` exception is raised.

Pattern Matching in OCaml (3/3)

- Pattern matching can be used to define a function by cases.
- The form

  `function` p_1 `->` e_1 `|` $\cdots$ `|` p_n `->` e_n

  is equivalent to

  `function` e `->` `match` e `with` p_1 `->` e_1 `|` $\cdots$ `|` p_n `->` e_n
- Patterns can be named: p `as` name
- Patterns can have conditions: p `when` cond
- Value declarations can use pattern matching:

```
let (x,y,z) = (1,1.,true) ;;
```