

Sequence ADTs

CIS*2520 Summer 2006

List ADT Specification

- DEF'N:
 - A list is a finite linear sequence of elements (of a given type T); for each element:
 1. either the element has a unique predecessor or has no predecessor and is the head of the list (which is unique)
 2. either the element has a unique successor or has no successor and is the tail of the list (which is unique)
- OPERATIONS:
 - **create**
 - create a new List initialized to be empty
 - PRE: n/a
 - POST: list is initialized and empty
 - **destroy**
 - destroy a list
 - PRE: n/a
 - POST: list is destroyed

CIS*2520 - Summer 2006

D. McCaughan

List ADT Specification (cont.)

- OPERATIONS (cont.):
 - **head**
 - adjust current position to the beginning of the list
 - PRE: list is not empty
 - POST: current element is now the first one
 - **tail**
 - adjust current position to the end of the list
 - PRE: list is not empty
 - POST: current element is now the last one
 - **prev**
 - adjust current position to the previous element in the list
 - PRE: list is not empty and not already at the beginning
 - POST: current element is now the previous one in sequence

CIS*2520 - Summer 2006

D. McCaughan

List ADT Specification (cont.)

- OPERATIONS (cont.):
 - **next**
 - adjust current position to the next element in the list
 - PRE: list is not empty and not already at the end
 - POST: current element is now the next one in sequence
 - **moveToNth**
 - adjust current position to the "Nth" element of the list (counted from 1)
 - PRE: $n \leq \text{size of list}$
 - POST: current element is now the "Nth" element
 - **isEmpty**
 - determine if list is empty
 - PRE: n/a
 - POST: evaluates to "true" if list is empty, "false" otherwise

CIS*2520 - Summer 2006

D. McCaughan

List ADT Specification (cont.)

- OPERATIONS (cont.):
 - *addAfter*
 - *addBefore*
 - *currentElement*
 - *setCurrentElement*
 - *removeElement*
 - *getLength*
 - *getPosition*
 - *write*
 - *read*

 - Others?

CIS*2520 - Summer 2006

D. McCaughan

List ADT Implementation Issues

- Types of lists:
 - linear
 - array-based
 - static/dynamic
 - linked
 - array-based
 - reference/pointer-based
 - single-linked
 - double-linked
 - circular
 - others?

CIS*2520 - Summer 2006

D. McCaughan

Applications of the List ADT

- Efficiency of Cellular Automata
 - **"Game of Life"** - John Conway
 - RULES:
 1. the neighbours of a cell are the eight cells that touch it vertically, horizontally or diagonally
 2. if a cell is alive but either has no neighbouring cells alive or only one alive, then in the next generation the cell dies of loneliness
 3. if a cell is alive and has four or more neighbouring cells also alive, then in the next generation the cell dies of overcrowding
 4. a living cell with either two or three living neighbours remains alive in the next generation
 5. if a cell is dead, then in the next generation it will become alive if it has exactly three neighbouring cells that are already alive; all other cells remain dead in the next generation
 6. all births and deaths take place at exactly the same time (dying cells cannot give birth to or support others, nor can newly alive cells kill or support others).

CIS*2520 - Summer 2006

D. McCaughan

The Game of Life

- Implementation options:
 - the obvious is to traverse a 2-D array doing counts
 - this is highly inefficient for sparse arrays
 - most cells tend to be empty --- waste of time traversing them all
 - alternative? Use lists!
 - list of "live" and "dead" cells
- General idea:
 - initialize array and compute initial neighbour counts
 - construct list of cells to become alive/dead in next generation

CIS*2520 - Summer 2006

D. McCaughan

The Game of Life (cont.)

- repeat the following:
 - for each cell on “to be made alive” list
 - make cell alive
 - update neighbour counts
 - if a neighbour count == value to become alive, add cell to list to be made alive in next generation (another list)
 - for each cell on “to be made dead” list
 - make cell dead
 - update neighbour counts
 - if neighbour count == value to become dead, add cell to list to be made dead in next generation (another list)
 - write out the array
 - copy “next generation” lists to “current generation” lists
- issues?

CIS*2520 - Summer 2006

D. McCaughan

The Game of Life (cont.)

1	2	2	1	live_next:	1	2	2	1
2	2	3	2	(3,1)	3	2	2	1
3	3	5	1	(4,3)	2	4	4	2
2	1	3	1	dead_next:	2	3	1	1
				(3,4)				
				(4,2)				

- Effect of using lists over the pure array?
 - for 100x100 grid:
 - array traversal requires approx. 80000 statements per iteration
 - list technique requires approx. 5000 statements per iteration
- See also: Game of Life example

CIS*2520 - Summer 2006

D. McCaughan

Extensions of the List ADT

- Queue, Stack, Set, etc.
- Consider: Polynomial Arithmetic
 - use Lists to store the coefficient/exponents of the terms in a polynomial
 - Polynomial ADT?
- How would you organize the implementation of these components?

CIS*2520 - Summer 2006

D. McCaughan

Queue ADT Specification

- **DEF’N:**
 - A queue is a list [a finite linear sequence of elements] in which all insertions occur at the end, and all deletions occur at the head (FIFO) :
- **OPERATIONS:**
 - **create**
 - create a new queue initialized to be empty
 - PRE: n/a
 - POST: queue is initialized and empty
 - **destroy**
 - destroy a queue
 - PRE: n/a
 - POST: queue is destroyed

CIS*2520 - Summer 2006

D. McCaughan

Queue ADT Specification (cont.)

- **OPERATIONS (cont.):**
 - **arrive**
 - add an entry to the end of the queue
 - PRE: n/a
 - POST: new entry in sequence as the last element in the queue
 - **depart**
 - removes an entry from the front of the queue
 - PRE: queue is not empty
 - POST: front entry in the queue is removed and returned; the front of the queue is now the deleted element's successor
 - **front**
 - examine the element currently at the front of the queue
 - PRE: queue is not empty
 - POST: returns the element from the front of the queue (element is not removed from queue)

CIS*2520 - Summer 2006

D. McCaughan

Queue ADT Specification (cont.)

- **OPERATIONS (cont.):**
 - **length**
 - obtain the current length of the queue
 - PRE: n/a
 - POST: returns the current length of the queue
 - **isEmpty**
 - determine if queue is empty
 - PRE: n/a
 - POST: evaluates to "true" if queue is empty, "false" otherwise
 - **write**
 - **read**
 - **others?**

CIS*2520 - Summer 2006

D. McCaughan

Queue ADT Impl. Issues

- Implement from scratch
 - array-based
 - linked
 - similar basic issues faced with the List
- But...a queue is very similar to a list
 - another option: encapsulate a List ADT
 - consider how each of these operations is implemented in terms of a List
 - ADT interface is straightforward
 - a List as a field in the structure/type
 - all functions implemented to provide a "queue-looking" interface to the component List

CIS*2520 - Summer 2006

D. McCaughan

Applications of the Queue ADT

- System Simulation
 - suppose we want to simulate an airport as follows:
 - a single runway
 - in each unit of time one plane can either take-off or land
 - there may be several planes waiting to take-off and land
 - landing planes have priority (planes may only take off if there are none waiting to land)
 - what do we need (in software) to realize this model?
 - time
 - landing queue, take-off queue
 - some means of generating planes
 - i.e. collect statistics, average runway idle time, etc.

CIS*2520 - Summer 2006

D. McCaughan

Airport Simulation

- planes wanting to land do
 - arrive(landing_queue, plane_id);
- planes wanting to take-off do
 - arrive(takeoff_queue, plane_id);

```
init. time to 0
init landing/takeoff queues

for each time step do:
  ...
  if landing queue not empty      if (!isEmpty(landing))
    allow next plane to land      depart(landing);
  else if takeoff queue not empty else if (!isEmpty(takeoff))
    allow next plane to takeoff    depart(takeoff);
  else
    runway is idle
  ...
```

CIS*2520 - Summer 2006

D. McCaughan

PriorityQueue ADT

- A queue with the notion of priorities
 - elements retrieved in order by highest to lowest priority, in order of arrival
 - i.e. hospital triage
- Specification?
- Implementation?
- Can we extend something to build this or we doing it from scratch?

CIS*2520 - Summer 2006

D. McCaughan

Stack ADT Specification

- DEF'N:
 - A stack is a list [a finite linear sequence of elements] in which all insertions and deletions occur at the head (LIFO)
- OPERATIONS:
 - **create**
 - create a new stack initialized to be empty
 - PRE: n/a
 - POST: stack is initialized and empty
 - **destroy**
 - destroy a stack
 - PRE: n/a
 - POST: stack is destroyed

CIS*2520 - Summer 2006

D. McCaughan

Stack ADT Specification (cont.)

- OPERATIONS (cont.):
 - **push**
 - place an entry onto the top of the stack
 - PRE: n/a
 - POST: new entry in sequence as the top (first) element in the stack
 - **pop**
 - remove an entry from the top of the stack
 - PRE: stack is not empty
 - POST: top (first) entry on the stack is removed and returned; the top of the stack is now the deleted element's successor
 - **peek**
 - examine the element currently on the top of the stack
 - PRE: stack is not empty
 - POST: returns the element from the top of the stack (element is not removed from stack)

CIS*2520 - Summer 2006

D. McCaughan

Stack ADT Specification (cont.)

- OPERATIONS (cont.):

- **depth**
 - obtain the current depth of the stack
 - PRE: n/a
 - POST: returns the current depth (size) of the stack
- **isEmpty**
 - determine if stack is empty
 - PRE: n/a
 - POST: evaluates to “true” if stack is empty, “false” otherwise
- **write**
- **read**
- **others?**

CIS*2520 - Summer 2006

D. McCaughan

Stack ADT Impl. Issues

- Consider how each of these operations would be:
 - implemented from scratch
 - implemented in terms of List operations (i.e. encapsulating a List)

```
typedef struct Stack_t {
    List *list;
} Stack;
...
createStack(void) {
    ...
    new->list = createList();
}

stackPush(Stack *s, void *data) {
    listHead(s->list);
    listAddBefore(s->list, data);
}
...

```

CIS*2520 - Summer 2006

D. McCaughan

Stack ADT Applications

- Memory Management
 - essentially a “double stack” - two logical stacks (or two “tops” superimposed on a single physical representation)
 - dynamic memory allocated from the heap
 - note: the heap isn’t really a stack, but is sufficient for this illustration
 - automatic allocation takes place on the stack
 - ISSUES?
 - How would you design/implement this?



CIS*2520 - Summer 2006

D. McCaughan

Stack ADT Applications (cont.)

- Reverse Polish Notation
 - INFIX notation (*op1 operator op2*) by itself is ambiguous
 - e.g. $2 - 1 + 3$ ($= 4?$ $= -2?$) --- need order of ops or parentheses
 - Jan Lukasiewicz (Polish logician)
 - noted parentheses not required if you use POSTFIX (*op1 op2 operator*)
 - read left to right, evaluate previous 2 operands when hit an operator
 - note: you should be able to do this

Infix	Postfix
$A + B$	$AB+$
$(A - B) * C$	$AB-C*$
$(A + B) / (B * A)$	$AB+BA*/$
$(A * (B + C)) / D$	$ABC+*D/$
$A * ((B - C) / (D + E))$	$BC-DE+/A*$

CIS*2520 - Summer 2006

D. McCaughan

Stack ADT Applications (cont.)

- Other thoughts...
 - reversing a string/numbers/any sequence
 - searching (backtracking)
 - e.g. Maze ADT (not a stack, but might use one)

CIS*2520 - Summer 2006

D. McCaughan

Explicit Implementation

- We have primarily focused on implementing stacks and queues by encapsulating a list rather than implementing them directly
- Depending on the application it may be desirable to implement a Stack or Queue ADT explicitly (or as a self-contained ADT rather than relying on a list)
 - in practice this is almost never the case (why?)
 - however:
 - may need a “special” type of queue or stack (e.g. DoubleStack)
 - language issues may force you out of the strict ADT mode

CIS*2520 - Summer 2006

D. McCaughan

A Step Backward

- It is often convenient (or necessary, i.e. COBOL, Fortran, etc.) to view list-type structures in the absence of pointers (or even ADTs for that matter)
- e.g. a stack implemented with an array:

```
#define MAX_SIZE ...
typedef Stack_t
{
    int top;
    void *val[MAX_SIZE];
} Stack;
...
stackPop(Stack *s)
{
    if (s->top > 0)
        return(s->val[s->top--]);
    else
        /* stack is empty */
}
...
```

CIS*2520 - Summer 2006

D. McCaughan

A Step Backward (cont.)

- Similar application to queues
- It is critical that you understand this conceptually
 - contiguous vs. linked
 - extend vs. direct implementation
 - all decisions should be *justified*, not random

```
...
stackPush(Stack *s, void *data)
{
    if (s->top < MAX_SIZE)
        s->val[++s->top] = data;
    else
        // stack is full
}
...
```

CIS*2520 - Summer 2006

D. McCaughan

Notes

- Always remember that the interface is King (Queen?)
- Always be thinking as if you have only a library and no access to the underlying implementation
 - better modularity will result naturally
- And finally...
 - how would you provide a new function for an ADT that you do not have the source code for?