Data Structures: Stacks, Queues and Lists

Algorithms

Elena Hernández Pereira, Santiago Jorge

Computer Science Department
Faculty of Computer Science

UNIVERSIDADE DA CORUÑA

- Weiss, M.A. Data structures and algorithms analysis. Benjaming/Cummings, 1995. **Chapter 3: Lists, stacks and queues (pages 41-86)**
- Brassard, G. and Bratley, P. Fundamentals of algorithmics. Prentice Hall, 1996. **Chapter 5: Data structures (pages 147-186)**
- Peña Marí, R. Diseño de Programas, Formalismo y Abstracción. Prentice Hall, 1998. **Capítulo 7: Implementación de estructuras de datos (pages 257-290)**

- The element deleted is always the one most recently inserted
- Implements a LIFO – Last In First Out – policy
- Basic operations are *push*, *pop* and *top*
    - A *top* or *pop* on an empty stack is an error in the stack ADT
    - Running out of space performing a *push* is an implementation error
- All the operations take constant time notwithstanding the number of elements pushed

Array implementation of Stacks

```
type
 Stack = record
   TopOfStack : [0..MaxStackSize];
   StackArray : array [1..MaxStackSize] of ElementType
 end record

procedure CreateStack (S)
  S.TopOfStack := 0
end procedure

function EmptyStack (S): boolean
  return S.TopOfStack = 0
end function
```

Array implementation of Stacks (II)

```
procedure Push (x,S)
  if S.TopOfStack = MaxStackSize then
    error "Full stack"
  else
    S.TopOfStack = S.TopOfStack + 1;
    S.StackArray [S.TopOfStack] := x
end procedure

function Top (S): ElementType
  if EmptyStack(S) then
    error "Empty stack"
  else
    return S.StackArray [S.TopOfStack]
end function

procedure Pop (S)
  if EmptyStack(S) then
    error "Empty stack"
  else
    S.TopOfStack = S.TopOfStack − 1
end procedure
```

Stack.h

```
#ifndef MaxStackSize
#define MaxStackSize 10
#endif

typedef int ElementType;
typedef struct {
  int top;
  ElementType array[MaxStackSize];
  } stack;

      void CreateStack(stack *);
       int EmptyStack(stack);
      void Push(ElementType, stack *);
ElementType Top(stack);
      void Pop(stack *);

/* ERRORS: Top or Pop over an empty stack
           Push over a full stack */
```

Stack.c

```
#include <stdlib.h>
#include <stdio.h>
#include "Stack.h"

void CreateStack(stack *S) {
  S->top = -1;
}

int EmptyStack(stack S) {
  return (S.top == -1);
}

void Push(ElementType x, stack *S) {
  if (++S->top == MaxStackSize) {
    printf("error: Full stack\n");
    exit(EXIT_FAILURE);
  }
  S->array[S->top] = x;
}
```

Stack.c (II)

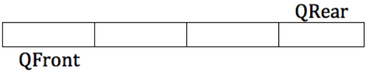
```
ElementType Top(stack S) {
  if (EmptyStack(S)) {
    printf("error: Empty stack \n");
    exit(EXIT_FAILURE);
  }
  return S.array[S.top];
}

void Pop(stack *S) {
  if (EmptyStack(*S)) {
    printf("error: Empty stack \n");
    exit(EXIT_FAILURE);
  }
  S->top--;
}
```

- The element deleted is always the former inserted
- Implements a FIFO – First In First Out – policy
- Basic operations are *enqueue*, *dequeue* and *front*
  - A *front* or *dequeue* on an empty queue is an error in the queue ADT
  - Running out of space performing an *enqueue* is an implementation error
- All the operations take constant time notwithstanding the number of elements enqueued

- The positions QFront and QRear represent the front and the end of the queue
- Whenever these positions get the end of the array, they are wrapped around to the beginning
- Status of the queue during some operations:

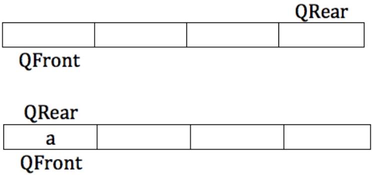
1) CreateQueue (Q)



- The positions QFront and QRear represent the front and the end of the queue
- Whenever these positions get the end of the array, they are wrapped around to the beginning
- Status of the queue during some operations:

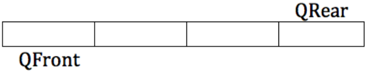
1) CreateQueue (Q)

2) EnQueue (a, Q)

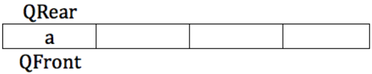


- The positions QFront and QRear represent the front and the end of the queue
- Whenever these positions get the end of the array, they are wrapped around to the beginning
- Status of the queue during some operations:

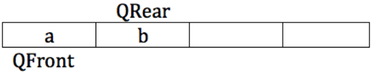
1) CreateQueue (Q)



2) EnQueue (a, Q)

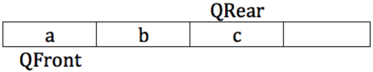


3) EnQueue (b, Q)



- Status of the queue during some operations:

4) EnQueue (c, Q)



- Status of the queue during some operations:

4) EnQueue (c, Q)

QRear

a	b	c	

QFront

5) EnQueue (d, Q)

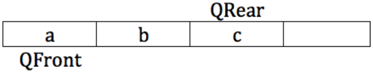
QRear

a	b	c	d

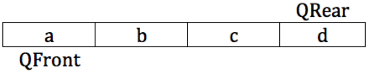
QFront

- Status of the queue during some operations:

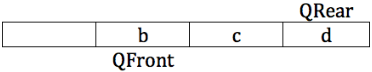
4) EnQueue (c, Q)



5) EnQueue (d, Q)



6) DeQueue (Q)



- Status of the queue during some operations:

4) EnQueue (c, Q)

		QRear	
a	b	c	
QFront			

5) EnQueue (d, Q)

			QRear
a	b	c	d
QFront			

6) DeQueue (Q)

			QRear
	b	c	d
	QFront		

7) EnQueue (e, Q)

QRear			
e	b	c	d
	QFront		

```
type
 Queue = record
   QFront, QRear : [0..MaxQSize];
   QSize : [0..MaxQSize];
   QArray : array [1..MaxQSize] of ElementType
 end record

procedure CreateQueue (Q)
  Q.QSize := 0;
  Q.QFront := 1;
  Q.QRear := MaxQSize
end procedure

function EmptyQueue (Q): boolean
  return Q.QSize = 0
end function
```

```
procedure Increment (x) /* private */
  if x = MaxQSize then
    x := 1
  else
    x := x + 1
end procedure

procedure Enqueue (x,Q)
  if Q.QSize = MaxQSize then
    error "Full Queue"
  else
    Q.QSize := Q.QSize + 1;
    Increment (Q.QRear);
    Q.QArray [Q.QRear] := x
end procedure
```

```
function Dequeue (Q): ElementType
  if EmptyQueue(Q) then
    error "Empty Queue"
  else
    Q.QSize := Q.QSize - 1;
    x :=  Q.QArray[Q.QFront];
    Increment(Q.QFront);
    return x
end function


function Front (Q) : ElementType
  if EmptyQueue(Q) then
    error "Empty Queue"
  else
    return Q.QArray[Q.QFront]
end function
```

Queue.h

```
#ifndef MaxQSize
#define MaxQSize 5
#endif

typedef int ElementType;
typedef struct {
  int QFront, QRear, QSize;
  ElementType array[MaxQSize];
  } queue;


void CreateQueue(queue *);
int EmptyQueue(queue);
void Enqueue(ElementType, queue *);
ElementType Dequeue(queue *);
ElementType Front(queue);


/* ERRORS: Dequeue or Front over an empty queue
           Enqueue over a full queue */
```

Queue.c

```
#include <stdlib.h>
#include <stdio.h>
#include "Queue.h"

void CreateQueue (Queue *Q) {
  Q->QSize  =  0;
  Q->QFront =  0;
  Q->QRear  = -1;
}

int EmptyQueue (Queue Q) {
  return (Q.QSize == 0);
}

void Increment (int *x) { /* private */
if (++(*x) == MaxQSize)
*x = 0;
}
```

Queue.c (II)

```
void Enqueue (ElementType x, Queue *Q) {
  if (Q->QSize == MaxQSize) {
    printf("error: Full Queue: %d\n", Q->QSize);
    exit(EXIT_FAILURE);
  }
  Q->QSize++;
  Increment (&(Q->QRear));
  Q->array[Q->QRear] = x;
}

ElementType Front (Queue Q) {
  if (EmptyQueue(Q)) {
    printf("error: Empty Queue\n");
    exit(EXIT_FAILURE);
  }
  return(Q.array[Q.QFront]);
}
```

Queue.c (III)

```
ElementType Dequeue (Queue *Q) {
  ElementType x;

  if (EmptyQueue(*Q)) {
    printf("error: EmptyQueue\n");
    exit(EXIT_FAILURE);
  }
  Q->QSize--;
  x = Q->array[Q->QFront];
  Increment(&(Q->QFront));
  return x;
}
```

Lists

- A list is a data structure consisting of a number of elements of the same type
- Basic operations:
  - `Print` its content
  - `Find` the position of the first occurrence of an element
  - `Insert` and Delete some element from some position
  - `FindKth` element, which returns the element of the indicated position

Simple array

- An array size has to be declared
  - Estimation of the maximum size is required
  - Wastes considerable space
- Computational complexity of the operations
  - `FindKth`, takes constant time
  - `Print` and `Find`, take linear time
  - `Insert` and `Delete` are expensive
    - Requires half the list to be moved for either operation, so linear time is required (O(n))
    - Building a list or deleting all its elements would require quadratic time

Linked

- Each node points to its successor. The last node points to *nil*
- The list points to the first node (and the last)
- Computational complexity of the operations:
    - `Print` and `Find` take linear time
    - `Delete` makes one pointer change and a dispose operation (O(1))
        - Uses `FindPrevious` with linear execution time
    - `Insert` after a position needs a `new` call and two pointer operations, (O(1))
        - `Find` the position should take linear time
    - Use a header node makes easier `Insert` and `Delete` at the beginning of the list

Doubly linked

- Each nodes points to the previous and the followings ones
- Doubles space memory needed for the pointers
- Doubles the cost of insertions and deletions pointer management
- Simplifies `Delete` operation
    - `FindPrevious` is no longer needed

Linked list with a header

```
type
  NodePtr  = ^ Node;
  List     = NodePtr;
  Position = NodePtr;

  Node = record
    Element : ElementType;
    Next    : NodePtr
  end record

procedure CreateList ( L )
  new ( tmp );
  if tmp = nil then
    error "Out of space"
  else
    tmp^.Element := {Header node };
    tmp^.Next := nil;
    L := tmp
end procedure
```

Linked list with a header (II)

```
function EmptyList ( L ) : boolean
  return L^.Next = nil
end function


function Find ( x, L ) : Position
  p := L^.Next;
  while  p <> nil and p^.Element <> x do
    p := p^.Next
  return p
end function


function IsLast ( p ) : boolean /* private */
  return p^.Next = nil
end function
```

Linked list with a header (III)

```
function FindPrevious ( x, L ) : Position /* private */
  p := L;
  while p^.Next <> nil and p^.Next^.Element <> x do
    p := p^.Next
  return p
end function


procedure Delete ( x, L )
  p := FindPrevious ( x, L );
  if  IsLast ( p ) then
    error "Not found"
  else
    tmp := p^.Next;
    p^.Next := tmp^.Next;
    dispose ( tmp )
end procedure
```

Linked list with a header (IV)

```
procedure Insert ( x, L, p ) /* After p position */
  new ( tmp );
  if tmp = nil then
    error "Out of space"
  else
    tmp^.Element := x;
    tmp^.Next := p^.Next;
    p^.Next := tmp
end procedure
```

List.h

```
struct node {
  void *elem; /* "void *" is a generic pointer  */
  struct node *next;
};

typedef struct node *position;
typedef struct node *list;

list CreateList ();
int IsEmptyList (list l);
void Insert (void *e, position p);
/*insert e after the node pointed by p*/

position Find (list l, void *e,
            int (*comp)(const void *x, const void *y));
/*this function returns a value greater, equal
  or smaller than 0 in case x should be greater, equal
  or smaller than y */
```

List.h

```
void Delete (list l, void *e,
      int (*comp)(const void *x, const void *y));

position First (list l);
position Next (position p);
int IsEndOfList (position p);
void *Element (position p);


  /* To go through the list:
  for (p=First(l); !IsLast(p); p=Next(p)) {
    // Do something with element(p)
  }
 */
```

List.c

```
#include <stdlib.h>
#include <stdio.h>
#include "List.h"

static struct node *CreateNode(){
  struct node *tmp = malloc(sizeof(struct node));

  if (tmp == NULL) {
    printf("Out of space \n");
    exit(EXIT_FAILURE);
  }
  return tmp;
}

list CreateList (){
  struct node *l = CreateNode ();

  l->Next = NULL;
  return l;
}
```

List.c (II)

```
int IsEmptyList (list l){
  return (l->Next == NULL);
}

void Insert (void *x, position p) {
  struct node *tmp = CreateNode ();
  tmp->Element = x;
  tmp->Next = p->Next;
  p->Next = tmp;
}

position Find (list l, void *e,
          int (*comp)(const void *x, const void *y)){
  struct node *p = l->Next;

  while (p != NULL && 0!=(*comp)(p->Element, e))
    p = p->Next;
  return p;
}
```

List.c (III)

```
static position FindPrevious(list l, void *x,
                int (*comp)(const void *, const void *)) {
  struct node *p = l;

  while (p->Next != NULL &&
          0!=(*comp)(p->Next->elem, x))
    p = p->Next;
  return p;
}

static int IsLast (struct node *p) {
  return (p->Next == NULL);
}
```

List.c (IV)

```
void Delete (list l, void *x,
             int (*comp)(const void *, const void *)) {

  struct node *tmp, *p = FindPrevious (l, x, comp);

  if (!IsLast (p)) {
    tmp = p->Next;
    p->Next = tmp->Next;
    free(tmp);
  }
}

position First (list l) { return l->Next; }

position Next(position p) { return p->Next; }

int IsEndOfList (position p) { return (p==NULL); }

void *Element(position p) {  return p->Element; }
```