

Stack and Queue



Summary

- ADT
- Stack
- Queue

2

ADT

All main programs rely on concept of *Abstract Data Type (ADT)*.
 ADT are useful to define algorithms in a more general way.

An ADT is a mathematical model on top of which a set of operations is defined.

3

Example

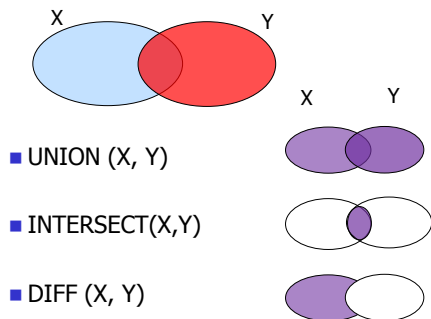
It is possible to define an ADT corresponding to a generic set of elements, on which some operations are defined, like *union*, *intersection* and *difference*:

- union (X, Y)
- intersect (X, Y)
- diff (X, Y)

These 3 operations have operands

4

Example of SET operations



5

ADT extends primitive types

- ADT concept combines and generalizes concepts of **primitive types** and **procedures**.
- Primitive types are the ones supported by a language (e.g. int, float, char).
- Using ADT it is possible to extend the set of types of supported data.
- An ADT is defined independently from:
 - The programming language
 - Implementation choices about data structures

6

Implementation in C

When implementing an ADT in C:

- Define a separate file containing the functions corresponding to the operations defined by the ADT
- Define a data type corresponding to the ADT
- Access the ADT only by means of its operations.

7

Example

Actual Type Definition:

```
typedef int SET_ELEM;
```

ADT operations:

```
SET_ELEM *union(SET_ELEM *, SET_ELEM *);
SET_ELEM *intersect(SET_ELEM *, SET_ELEM *);
SET_ELEM *diff (SET_ELEM *, SET_ELEM *);
SET_ELEM *make_null( void);
int size (SET_ELEM *);
void dump (SET_ELEM *);
```

8

Stack

A STACK is an ADT with these operations:

- **Push**: insert a new element in the ADT
- **Pop**: extract from ADT the last element inserted (LIFO = Last In First Out policy)
- **Empty**: return true if the stack has no elements
- **Init**: create an empty ADT.

9

Example

Suppose we execute these operations on a stack:

```
M = init()
Push (M, K1)
Push (M, K2)
Push (M, K3)
```

If `Pop (M)` is called it returns K3. Then another call to `Pop (M)` will return K2.

Calling then `Empty (M)` it returns FALSE.

10

Implementing stack with array

- Define an array of N elements
- Add a variable `top` containing the index of last inserted element.

Array memory is allocated in advance, whatever is the number of elements stored.

All operations on a stack have complexity $O(1)$.

11

Pseudo-code

```
STACK-EMPTY(S)
1  if top[S] = 0
2  then return TRUE
3  else return FALSE

PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x

POP(S)
1  if STACK-EMPTY(S)
2  then error "underflow"
3  else top[S] ← top[S] - 1
4  return S[top[S] + 1]
```

12

C

```

#define MAX 100
int buff[MAX];
int index;
void push( int val);
int pop( void);
int empty( void);
void push( int val)
{
  buff[index++] = val;
  return;
}

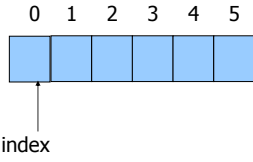
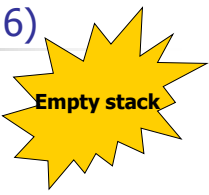
int pop( void)
{
  return( buff[--index]);
}
int empty( void)
{
  if( index == 0)
    return (1);
  else
    return (0);
}

```

13

Stack Example (n=6)

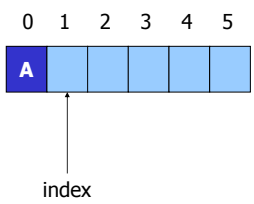
Init



14

Example (2)

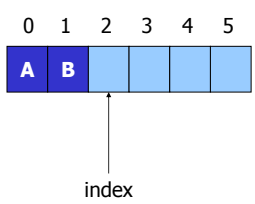
Push(S, A)



15

Example (3)

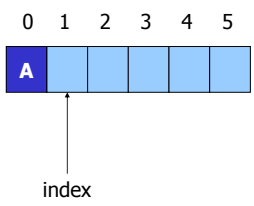
Push(S, B)



16

Example (4)

Pop(S) == B



17

Queue

- A queue is an ADT with these operations:
- **Insert** (or enqueue): insert a new element in ADT
 - **Extract** (or dequeue): extract from ADT the "oldest" element; it realizes a FIFO (*First In First Out*) strategy.
 - **Empty**: retrun true if ADT has 0 elements
 - **Init**: create an empty ADT.

18

Example

Suppose we have executed these operations:

```
M = init()
Enqueue (M, K1)
Enqueue (M, K2)
Enqueue (M, K3)
```

If Dequeue (M) is executed this returns K1.
 Executing again Dequeue (M), this returns K2.
 Calling Empty (M) at this point, it returns FALSE.

19

Implementing queue with array

- Definition of an array of $N+1$ elements
- Introduction of 2 variables head e tail:
 - Head contains the index of the oldest element, the one inserted for more time
 - Tail contains index in which insert a new element.
- A mechanism for transforming an array in a circular buffer.

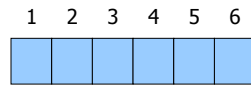
20

Circular Buffer

Head and tail are incremented whenever dequeue ed enqueue are called, respectively.
 Whenever head or tail become equals to $n+1$, they have to be reset at 1.
 At the beginning, head = 1, tail = 1.
 When the queue is full (overflow) then head = tail + 1 (or head=1 and tail=n).
 When the queue is empty (underflow) head = tail.

21

Example (n=5)

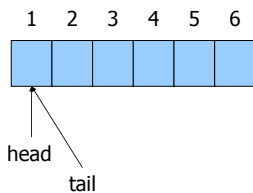


22

Example (1)

Init

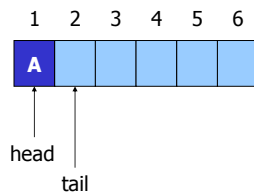
Empty queue



23

Example (2)

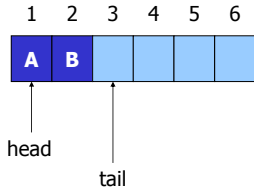
Enqueue(Q, A)



24

Example (3)

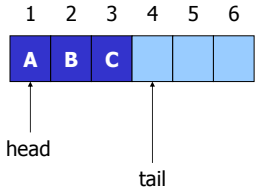
Enqueue(Q, B)



25

Example (4)

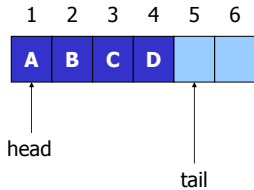
Enqueue(Q, C)



26

Example (5)

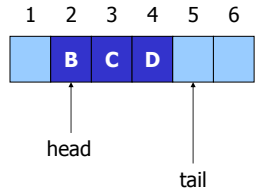
Enqueue(Q, D)



27

Example (6)

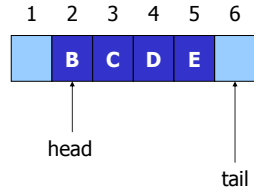
Dequeue(Q)==A



28

Example (7)

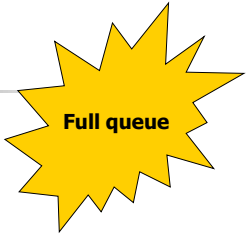
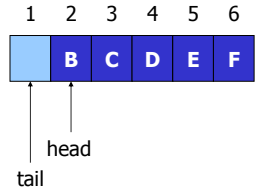
Enqueue(Q, E)



29

Example (8)

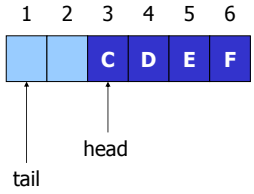
Enqueue(Q, F)



30

Example (9)

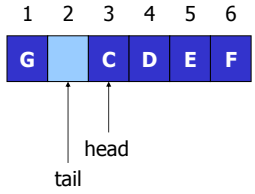
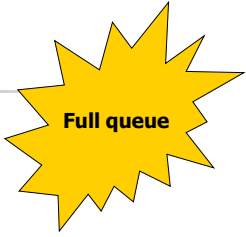
Dequeue(Q) == B



31

Example (10)

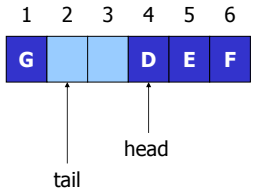
Enqueue(Q, G)



32

Example (11)

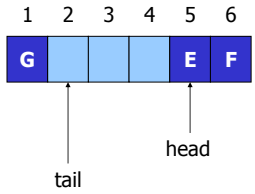
Dequeue(Q) == C



33

Example (12)

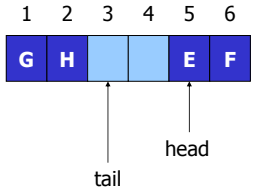
Dequeue(Q)



34

Example (13)

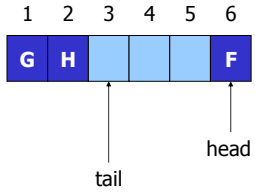
Enqueue(Q, H)



35

Example (14)

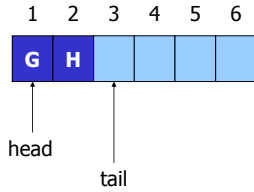
Dequeue(Q)



36

Example (15)

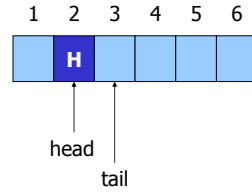
Dequeue(Q)



37

Example (16)

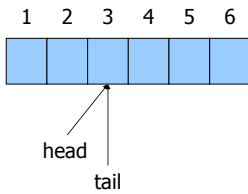
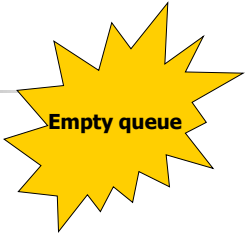
Dequeue(Q)



38

Example (17)

Dequeue(Q)



39

Pseudo-code

```

ENQUEUE (Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3    then tail[Q] ← 1
4    else tail[Q] ← tail[Q] + 1

DEQUEUE (Q)
1  x ← Q[head [Q]]
2  if head [Q] = head [Q]
3    then head [Q] ← 1
4    else head [Q] ← head [Q] + 1
5  return x
    
```

40

C

```

#define DIM 10
int  buffer[DIM+1];
int  tail=0;
int  head=0;
int  insert( int elem);
int  extract( void);
int  empty( void);

int insert( int elem)
{
    if( (tail+1) < head ||
        (tail==DIM && head==0) )
        return( -1);
    buffer[tail++] = elem;
    if( tail == (DIM+1))
        tail = 0;
    return( 0);
}
    
```

41

C

```

int extract( void)
{ int ret;
  if( head == tail)
    return( -1);
  ret = buffer[head++];
  if( head == (DIM+1))
    head = 0;
  return( ret);
}

int empty( void)
{
    if( head == tail)
        return( 1);
    else
        return( 0);
}
    
```

42