Linear ADT-1: Restricted Lists
Stacks, Queues

Atul Gupta

Linear ADTs

Restricted Lists
- Stack
- Queue
  - Circular queue
  - Priority queue

General Lists
- Arrays
- Linked list
- Circular list
- Doubly linked list

Stacks



Using Stacks: Checking Expression Validity

- Consider the following expression

[a + b *({c/d(1-n) + e/f (1+n)})/(g – sqrt[(b**2) – 4*a*c]/2)]

Using Stacks: Checking Expression Validity

```
bool ckeck_validity (char[ ] exp) {
        char next_char, popped_char;
        stack  S;
        bool valid = true;
         while (not_empty(exp)) {
            next_char  = get_next(exp);
            if ((next_char == '(') or (next_char =='{) or (next_char == '[')) then
                  push (S, next_char);
            if ((next_char == ')') or (next_char =='}) or (next_char == ']')) then {
                  popped_char  = pop (S);
                  if (next_char <> popped_char)
                       return (valid = false);
               }
        }
        If (not_empty(S)) then
             valid = false;
        return valid;
}
```

Stack

- Linear LIFO organization
- An attribute **top** always pointing to the most recently inserted data item
- Basic operations
  - void push(x, S) : Insert element x into S
  - item pop(S) : Return the last element inserted into S
  - boolean isStackEmpty(S): Return yes if S is empty

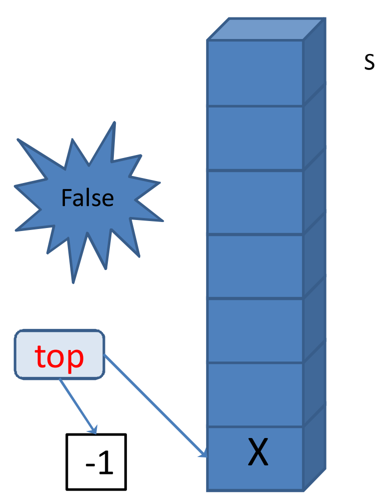
Applications of Stacks

- Direct applications
  - Delimiter matching
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Expression evaluations
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

7

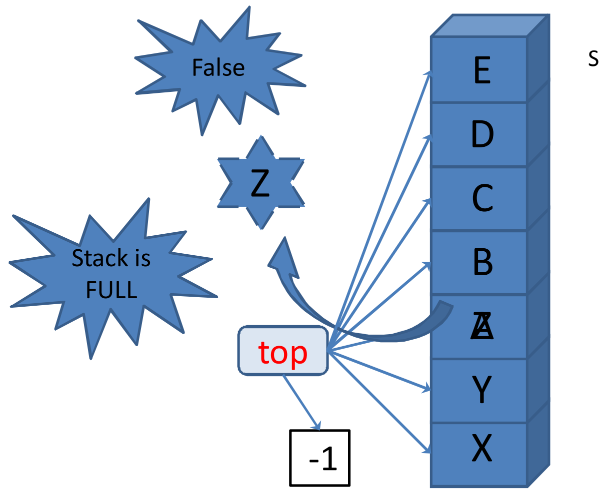
Stack: An Array Implementation

```
createStack(S): Define an array S
                for some fixed size N,
                top ← -1

push(x,S):  if top = N-1 then error
            else top ←top + 1
            S[top] ← x

isStackEmpty(S):  return (top < 0)

pop(S):  if isStackEmpty() then error
         else item ←S[top]
         top ← top - 1
         return (item)
```



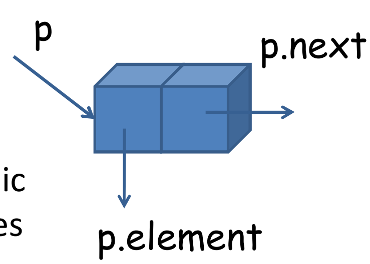
Stack Animation: Array Implementation

createStack()
push (X, S)
push (Y, S)
push (Z, S)
pop (S)
iStackEmpty (S)
push (A, S)
push (B, S)
push (C, S)
push (D, S)
push (E, S)
push (F, S)
...



Stack: Pointer Implementation

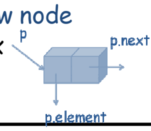
- Pointer facilitate dynamic implementation of a data structure
- Data is organized in a dynamic structure comprising of nodes where each node
  - is identified by a reference (pointer)
  - contains a data element
  - may points to next node/null



Stack: Pointer Implementation

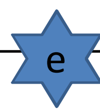
createStack(S): Define a pointer top,
top ← null

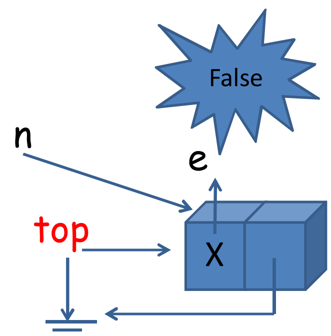
push(x,S): n = create new node
n.element ← x
n.next ← top
top ← n



isStackEmpty (S): return (top == null)

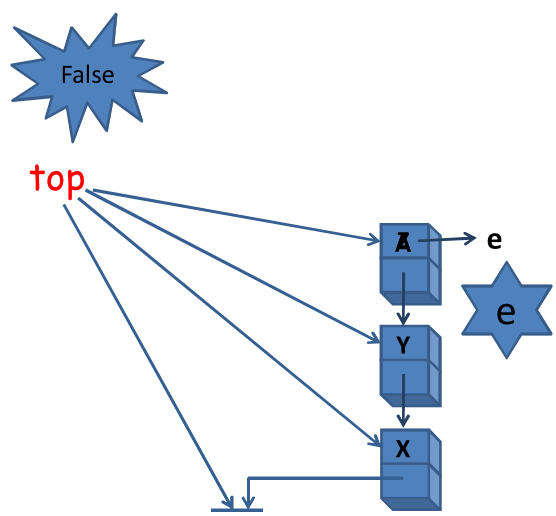
pop(S): if isStackEmpty() then error
else e ← top.element
top ← top.next
return (e)





Stack Animation: Pointer Implementation

createStack(S)
push (X, S)
push (Y, S)
push (Z, S)
pop (S)
iStackEmpty (S)
push (A, S)
push (B, S)
push (C, S)
push (D, S)
push (E, S)
push (F, S)
...



Stack: The running time

```
createStack(S): Define a pointer top,
                top ← null

push(x,S):  n = create new node
            n.element ←x
            n.next ← top
            top ← n

isStackEmpty (S): return (top == null)

pop(S):  if isStackEmpty() then error
         else e ←top.element
              top ← top.next
              return (e)
```

- Each operation takes O(1) time

Stack: C implementation using Array

```c
#define MAX 10
struct stack {
        int arr[MAX];
        int top;
} STACK;
void createStack (STACK *);
int stackEmpty ( STACK *)
int stackFull ( STACK *)
void push (STACK *, int item);
int pop (STACK *);
```

```c
void createStack (STACK * s ) {
   s -> top = -1;
}
```

```c
void push ( STACK *s, int item ) {
   if ( stackFull(s)) {
       printf ( "\nStack is full." );
       return;
   }
   s -> top++ ;
   s -> arr[s ->top] = item;
}
```

```c
int stackFull ( STACK *s) {
   if ( s -> top == MAX - 1 )
       return (1);
   else return (0);
}
```

```c
int stackEmpty ( STACK *s) {
   if ( s -> top == -1 )
       return (1);
   else return (0);
}
```

```c
int pop( struct stack *s ) {
   int item;
   if ( stackEmpty(s)) {
     printf ( "\nStack is empty." );
     return NULL;
   }
   item = s -> arr[s -> top];
   s -> top--;
   return item;
}
```

Stack: C implementation using Pointer

```
struct node {
        int data ;
        struct node *link ;
} ;
void createStack (struct node ** )
void push ( struct node **, int );
int pop ( struct node ** );
void delStack ( struct node ** );
Int stackEmpty ( struct node **);
```

```
int stackEmpty ( struct node **tos) {
   return ( *tos == NULL );
}
```

```
void createStack (struct node ** top ) {
    top = NULL;
}
```

```
void push ( struct node **top, int item ) {
   struct node *temp;
   temp = (struct node*) malloc(sizeof (struct node ));
   if ( temp == NULL )
         printf( "\nStack is full." );
   temp -> data = item;
   temp -> link = *top;
   *top = temp;
}
```

```
int pop ( struct node **top ) {
   struct node *temp;
   int item;
   if (stackEmpty(top)) {
         printf( "\nStack is empty." );
         return 0 ;
   }
   temp = *top;
   item = temp -> data;
   *top = ( *top ) -> link;
   free ( temp );
   return item;
}
```

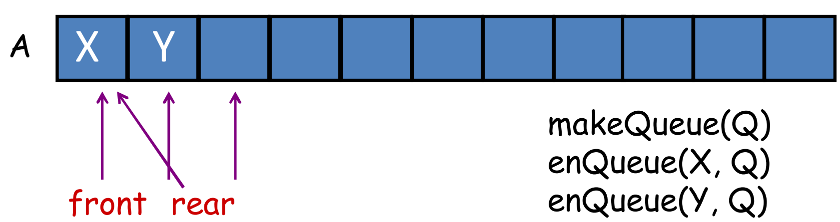
Queue

- Linear FIFO organization
- An attribute **rear** points to the place the next data item to be inserted
- An attribute **front** points to the next data item to be removed
- Basic operations
  - void enQueue(x,Q) : Insert element x into Q
  - item deQueue(Q) : Return the last element inserted into Q
  - boolean isQueueEmpty(Q): Return yes if Q is empty

Queue

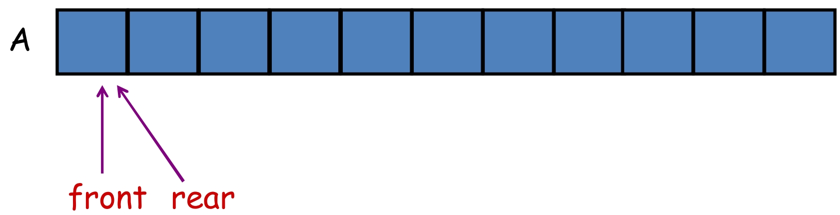
- void enQueue(x, Q) : Insert last element x into Q
- item deQueue(Q) : Delete the first element in Q
- boolean isQueueEmpty?(Q): Return yes if Q is empty
- item front(Q): Return the first element in Q
- int size(Q): Return the number of elements in the Q
- Queue make-queue(): Initialize a Q

Queue: Array Implementation



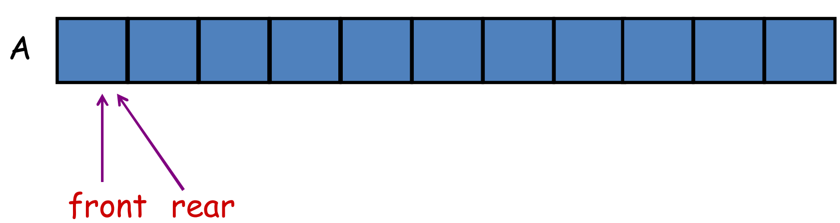
makeQueue(Q)
enQueue(X, Q)
enQueue(Y, Q)
enQueue(Z, Q)
deQueue(Q)
isQueueEmpty(
Q)
deQueue(Q)
enQueue(A, Q)
enQueue(B, Q)
…

Queue: Array Implementation



Empty Queue? front == rear

Circular Queue: Array Implementation



Empty Queue? front == rear

Full Queue? front == (rear+1) mod N + 1

Circular Queues: Types of Array Implementations

- front is always at first position
- An array with two indices always increasing
- A circular array with front and rear and one position is left vacant
- A circular array with two indices and a boolean variable
- A circular array with two indices and a count (size) variable
- Special values for array indices

Circular Array with size() function

```
isQueueEmpty(Q):
  return (front == rear)
```

```
front(Q):
  if (isQueueEmpty(Q)) then error
    else return A[front]
```

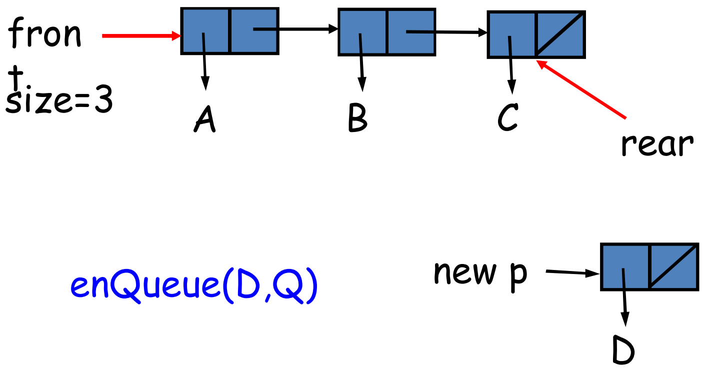
```
size(Q):
  if (rear >= front) then
          return (rear-front+1)
  else return (N-(front-rear))
```

```
enQueue(x,Q): if size(Q) == N then error
      else A[rear] ← x
      rear ← (rear+1) mod N
```

```
pop(Q):
if (isQueueempty(Q)) then error
        else e ←A[front]
      front ← (front + 1) mod N
       return (e)
```

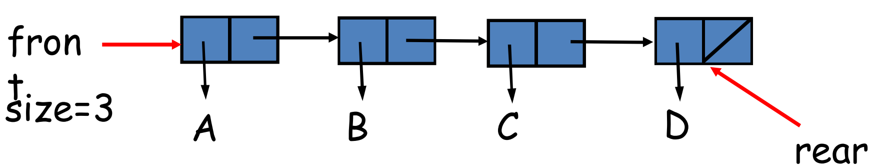
```
deQueue(Q):
if (isQueueempty?(Q)) then error
    else e ←A[front]
    front ← (front + 1) mod N
    return (e)
```

Queue: Pointer Implementation



enQueue(D,Q)

Implementation with lists



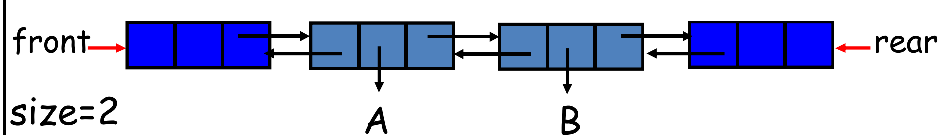
enQueue(D,Q)

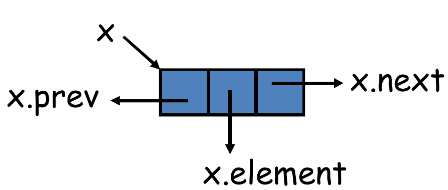
Double Ended Queue (**Deque**)

insertFront(x,D) : Insert x as the first in D
deQueue(D) : Delete the first element of D
enQueue(x,D): Insert x as the last in D
removeLast(D): Delete the last element of D
Size(D)
isQueueEmpty?(D)
make-deque()

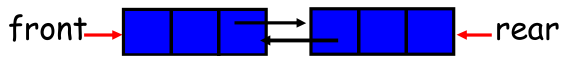
Implementation of Deque with Doubly Linked Lists

We use two sentinels (dummy nodes) here to make the code simpler





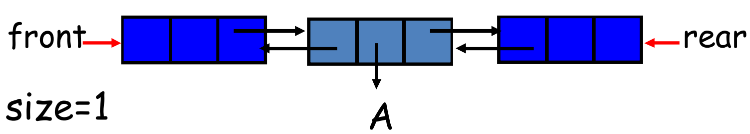
Empty Deque?



size=0

front.next ==rear && rear.prev == front

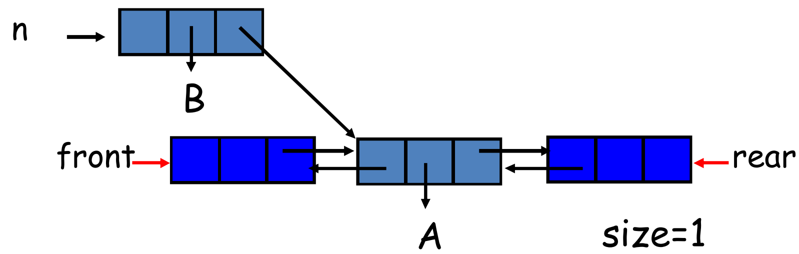
Insert Element in the Deque



size=1

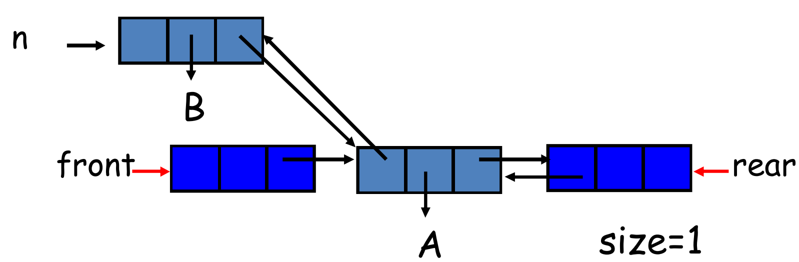
insertFront(B,D):

n = new node
n.element ←B
n.next ← front.next
(front.next).prev ← n
front.next ← n
n.prev← front
size ← size + 1



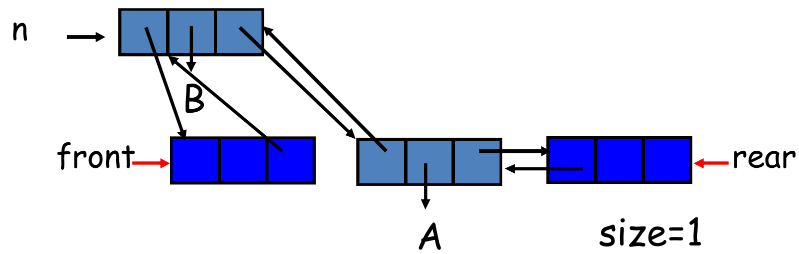
insertFront(B,D):

n = new node
n.element ← B
n.next ← front.next
(front.next).prev ← n
front.next ← n
n.prev ← front
size ← size + 1



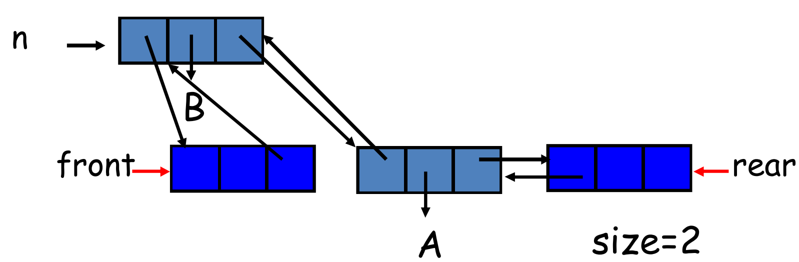
insertFront(B,D):

n = new node
n.element ← B
n.next ← front.next
(front.next).prev ← n
front.next ← n
n.prev ← front
size ← size + 1



insertFront(B,D):

n = new node
n.element ←B
n.next ← front.next
(front.next).prev ← n
front.next ← n
n.prev← front
size ← size + 1



insertFront(B,D):

n = new node
n.element ←B
n.next ← front.next
(front.next).prev ← n
front.next ← n
n.prev← front
size ← size + 1

Restricted Linear ADT: Summary

- Two important forms of linear restricted ADT are Stacks and Queues
- Stack is LIFO and Queue is FIFO
- All operations are O(1)
- Typically not used as search data structures
- Where to use them?