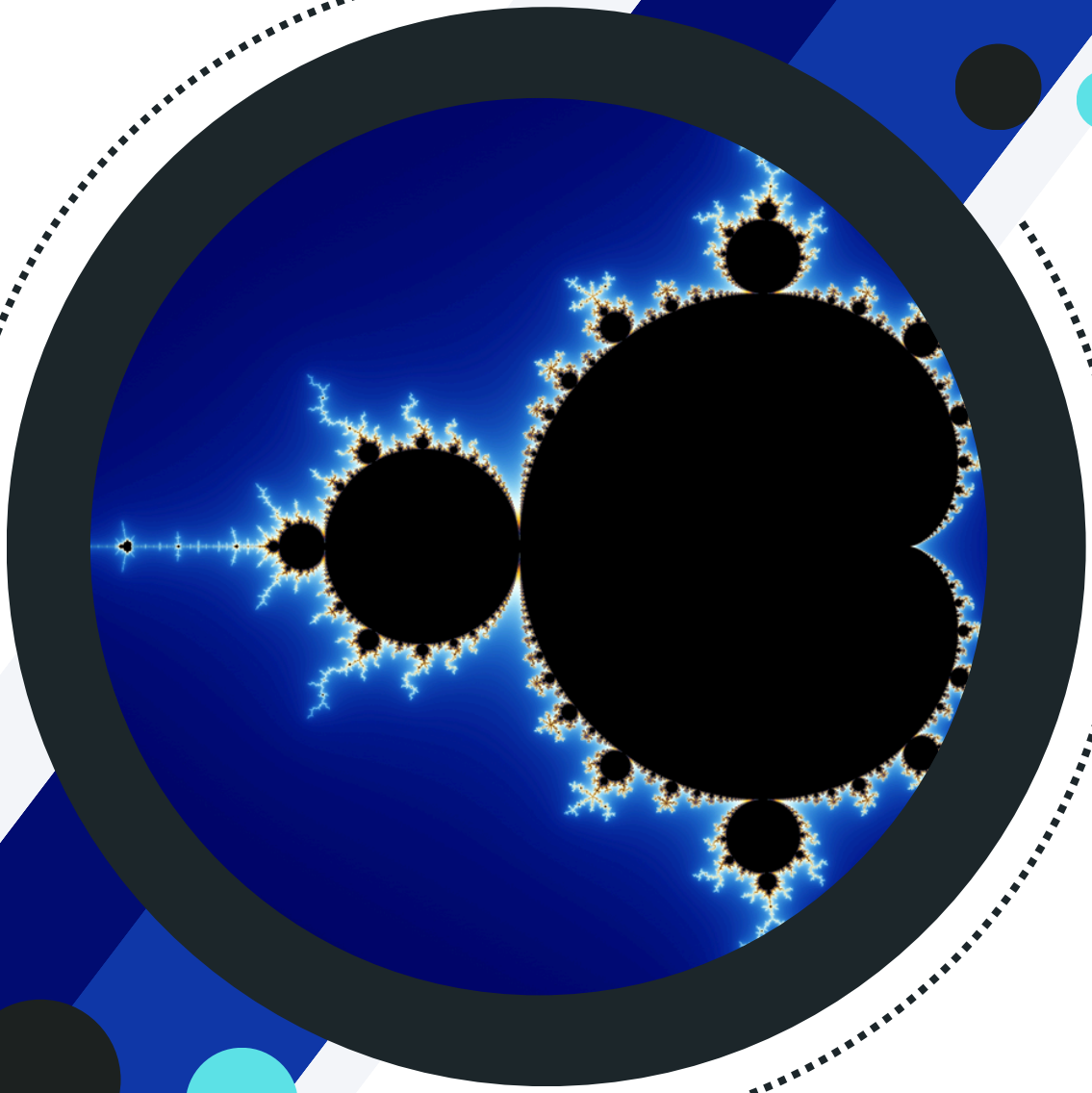
DEQUE PROJECT

WITH MIPS ASSEMBLY



Under supervision

Prof : Lamiaa Elrefaie

ELECTRICAL ENGINEERING
COMMUNICATION AND COMPUTER
ENGINEERING
CCE 307-ELE251 – COURSE PROJECT
(TERM 242)

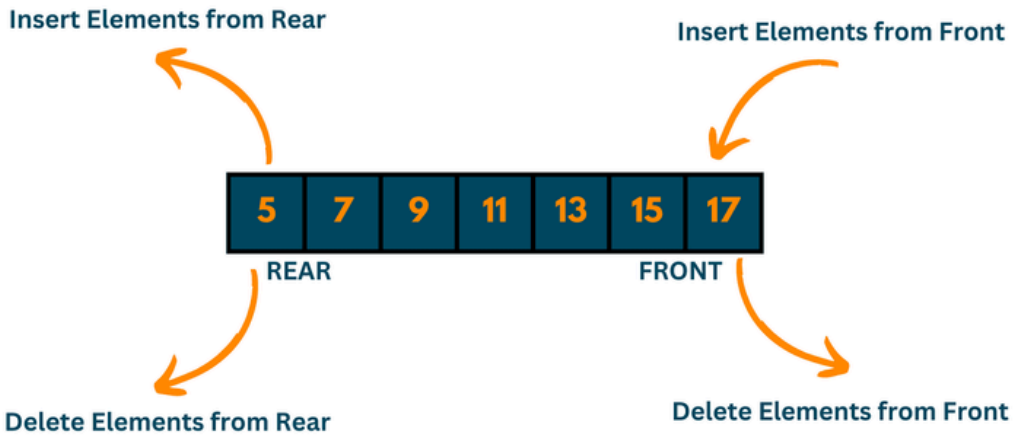
Course Project cover page

S#	Student Name	Edu Email	Student ID	Marks			
				Report & Slides (30)	Implementation (50)	Presentation (20)	Total (100)
1	Essam Eldin Hisham		231903704				
2	Mohamed Essam Ali		231903822				
3	Abdelrhman Emad Ibrahim		231903709				
4	Abdelrhman Tarek		221902993				
5	Mohamed Ahmed Amin		231903591				
6	Mostafa salah refaey		231903613				

Date handed in: 5 / 5 2024

TABLE OF CONTENTS:

Introduction



A deque, short for "double-ended queue," is a versatile data structure that allows insertion and deletion of elements from both ends. It provides functionality similar to both stacks and queues, enabling efficient operations like insertion and deletion at both the front and back of the deque. This versatility makes deques suitable for a wide range of applications where fast insertion and deletion at both ends are required.

The code

deque_mips.asm*

```
1   .data
2   deque: .space 40
3   frontIdx: .word -1
4   rearIdx: .word 10
5   MAX_SIZE: .word 10
6   used: .word 0
7   error_full_message: .asciiz "Deque is Full :(¥n"
8   error_empty_message: .asciiz "Deque is Empty :(¥n"
9   space: .asciiz " "
10
11  newline: .asciiz "¥n"
12  .text
13  .globl main
14
15  main:
16
17    li $v0, 10
18    syscall
19
20
21  Empty_error :
22    li $v0, 4
23    la $a0, error_empty_message
24    syscall
25
26    jr $ra
27
28  Full_error :
29    li $v0, 4
30    la $a0, error_full_message
31    syscall
32
33    jr $ra
34
```

deque_mips.asm*

```
35  push_front:
36      lw $t1, frontIdx
37      addi $t1 , $t1 , 1
38      lw $t2, MAX_SIZE
39      lw $t3, used
40      beq $t3, $t2, Full_error
41      la $t4, deque
42      sll $t5, $t1, 2
43      add $t4, $t4, $t5
44      sw $a0, 0($t4)
45      sw $t1, frontIdx
46      addi $t3, $t3, 1
47      sw $t3, used
48
49      jr $ra
50
51  push_back:
52      lw $t1, rearIdx
53      lw $t2, MAX_SIZE
54      lw $t3, used
55      addi $t1, $t1, -1
56      beq $t3, $t2, Full_error
57      la $t4, deque
58      sll $t5, $t1, 2
59      add $t4, $t4, $t5
60      sw $a0, 0($t4)
61      sw $t1, rearIdx
62      addi $t3, $t3, 1
63      sw $t3, used
64
65      jr $ra
```

```
67  pop_front:
68      lw $t1, frontIdx
69      lw $t2, used
70      beq $t2, $zero, Empty_error
71      addi $t1, $t1, -1
72      sw $t1, frontIdx
73      addi $t2, $t2, -1
74      sw $t2, used
75
76      jr $ra
77
78  pop_back:
79
80      lw $t1, rearIdx
81      lw $t2, used
82      beq $t2, $zero, Empty_error
83      addi $t1, $t1, 1
84      sw $t1, rearIdx
85      addi $t2, $t2, -1
86      sw $t2, used
87
88      jr $ra
```

deque_mips.asm*

```
display:
    lw $t1, frontIdx
    lw $t2, rearIdx
    la $t4 ,  deque
    displayfrontloop:
        bltz $t1 , exitedisplayfront
        sll $t5 , $t1 , 2
        add $t5 , $t4 , $t5
        lw $a0 , 0($t5)
         li $v0, 1
        syscall

        li $v0, 4
        la $a0, space
        syscall
        addi $t1  , $t1, -1
        j displayfrontloop


    exitedisplayfront:
    # print the back
    lw $t1 , MAX_SIZE
        addi $t1  , $t1, -1
    displayback:
        blt $t1 , $t2 , exitedisplayback
        sll $t5 , $t1 , 2
        add $t5 , $t4 , $t5
        lw $a0 , 0($t5)
         li $v0, 1
        syscall

        li $v0, 11
        li $a0, ' '
        syscall
        addi $t1  , $t1, -1
         j displayback

     exitedisplayback:

        li $v0 , 4
        la $a0 , newline
        syscall
        jr $ra
```

3. Explanation of the code

The Algorithm/implementation idea :

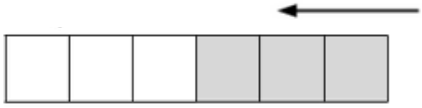
Output restricted double ended queue

This deque implementation employs two index pointers, frontIdx and rearIdx, to manage the front and rear positions of the deque, respectively.

When elements are pushed to front we add the value at arr[frontIdx] then we increment the frontIdx by 1

When elements are pushed to back we add the value at arr[rearIdx] then we decrement the rearIdx by 1

print the front elements



print the back elements

Printing the deque involves iterating through its elements. First, it prints elements from index = frontidx – 1 until reaching 0, then increments the index. Afterward, it prints elements from index 9 (assuming a maximum size of 10) until reaching rearIdx, then decrements the index.

This approach ensures that elements are inserted and printed in the correct order, maintaining the integrity and functionality of the deque.

The main Structure :

1. deque: This variable represents the deque data structure, which is initialized with a space of 40 bytes, implying a maximum capacity of 10 elements where each element occupies 4 bytes (assuming 32-bit integers).
2. frontIdx: This variable holds the index of the front element of the deque.
3. rearIdx: This variable holds the index of the rear element of the deque.
4. MAX_SIZE: This constant represents the maximum capacity of the deque.
5. used: This variable keeps track of the number of elements currently stored in the deque.
6. error_full_message: This string stores the error message to be displayed when attempting to perform an operation on a full deque.
7. error_empty_message: This string stores the error message to be displayed when attempting to perform an operation on an empty deque.

Push_front

C++ code

```cpp
void push_front(int val){
    if (used == SIZE) {
        cout << "Deque is Full :( \n" ;
        return;
    }
    deque[front] = val ;
    front = front - 1 ;
    used ++ ;
}
```

Mips code

```
35  push_front:
36      lw $t1, frontIdx
37      addi $t1 , $t1 , 1
38      lw $t2, MAX_SIZE
39      lw $t3, used
40      beq $t3, $t2, Full_error
41      la $t4, deque
42      sll $t5, $t1, 2
43      add $t4, $t4, $t5
44      sw $a0, 0($t4)
45      sw $t1, frontIdx
46      addi $t3, $t3, 1
47      sw $t3, used
48
49      jr $ra
```

Load registers:

$t1: Load the current value of frontIdx.

$t2: Load the maximum size of the deque (MAX_SIZE).

$t3: Load the current number of used elements (used).

Check if deque is full:

Compare the number of used elements ($t3) with the maximum size ($t2). If they are equal, jump to Full_error to handle the error.

Calculate memory address:

Load the base address of the deque into register $t4.

Shift the index ($t1) left by 2 bits (equivalent to multiplying by 4 since each element occupies 4 bytes).

Add the shifted index to the base address to get the memory address where the new element will be stored.

Store the value:

Store the value of the new element ($a0) into the calculated memory address.

Update front index:

Increment the front index ($t1) to reflect the insertion of a new element at the front.

Store the updated front index back into memory (frontIdx).

Update number of used elements:

Increment the count of used elements ($t3) to reflect the addition of a new element.

Store the updated count back into memory (used).

Return:

Return control to the calling function using jr $ra.

Push_front

push_front: This function inserts an element at the front of the deque.
It first checks if the deque is full, then inserts the element and updates the front index and the number of used elements accordingly.

```
13 main:
14    li $a0,5
15    jal push_front #content is [ 5 ]
16    li $a0,5
17    jal push_front #content is [ 5 5 ]
18    li $a0,5
19    jal push_front #content is [ 5 5 5 ] 3 x 5
20    li $a0,5
21    jal push_front #content is [ 5 5 5 5 ]
22    li $a0,5
23    jal push_front #content is [ 5 5 5 5 5 ] 5 x 5
24    li $a0,5
25    jal push_front #content is [ 5 5 5 5 5 5 ]
26    li $a0,5
27    jal push_front #content is [ 5 5 5 5 5 5 5 ] 7 x 5
28    li $a0,5
29    jal push_front #content is [ 5 5 5 5 5 5 5 5 ]
30    li $a0,5
31    jal push_front #content is [ 5 5 5 5 5 5 5 5 5 ] 8 x 5
32    li $a0,5
33    jal push_front #content is [ 5 5 5 5 5 5 5 5 5 5 ]
34    li $a0,5
35    jal push_front #content is [ 5 5 5 5 5 5 5 5 5 5 5 ]  10 x 5 FULL
36    li $a0,5
37    jal push_front #content is [ OVER FLOW (FULL array ) ]
38
```

```
Deque is Full :(

-- program is finished running --
```

here we push 10 elements in the deque and when we try to push another elements the error message showed

```
12
13 main:
14    li $a0,10
15    jal push_front #content is [ 10 ]
16
17    li $a0,7
18    jal push_front #content is [ 7 , 10 ]
19
20    li $a0,2
21    jal push_front #content is [ 2 , 7 , 10 ]
22
23    jal display
24    li $v0, 10
25    syscall
26
```

```
-- program is finished running --

2,7,10,
-- program is finished running --
```

here we push_front 3 elements and this is the output

Push_back

C++ code

```cpp
void push_back(int val){
    if (used == SIZE) {
        cout << "Deque is Full :( \n" ;
        return;
    }
    deque[back] = val ;
    back = back + 1 ;
    used ++ ;
}
```

Mips code

```
51  push_back:
52      lw $t1, rearIdx
53      lw $t2, MAX_SIZE
54      lw $t3, used
55      addi $t1, $t1, -1
56      beq $t3, $t2, Full_error
57      la $t4, deque
58      sll $t5, $t1, 2
59      add $t4, $t4, $t5
60      sw $a0, 0($t4)
61      sw $t1, rearIdx
62      addi $t3, $t3, 1
63      sw $t3, used
64
65      jr $ra
```

Load registers:

$t1: Load the current value of rearIdx.

$t2: Load the maximum size of the deque (MAX_SIZE).

$t3: Load the current number of used elements (used).

Check if deque is full:

Compare the number of used elements ($t3) with the maximum size ($t2). If they are equal, jump to Full_error to handle the error.

Calculate memory address:

Load the base address of the deque into register $t4.

Shift the index ($t1) left by 2 bits (equivalent to multiplying by 4 since each element occupies 4 bytes).

Add the shifted index to the base address to get the memory address where the new element will be stored.

Store the value:

Store the value of the new element ($a0) into the calculated memory address.

Update rear index:

Decrement the rear index ($t1) to reflect the insertion of a new element at the back.

Store the updated rear index back into memory (rearIdx).

Update number of used elements:

Increment the count of used elements ($t3) to reflect the addition of a new element.

Store the updated count back into memory (used).

Return:

Return control to the calling function using jr $ra.

Push_back

push_back: This function inserts an element at the back of the deque.
It first checks if the deque is full, then inserts the element and updates the front index and the number of used elements accordingly.

```
12
13  main:
14    li $a0,1
15    jal push_back #content is [ 1 ]
16    li $a0,2
17    jal push_back #content is [ 1 2]
18    li $a0,3
19    jal push_back #content is [ 1 2 3]
20    li $a0,4
21    jal push_back #content is [ 1 2 3 4]
22    li $a0,5
23    jal push_back #content is [ 1 2 3 4 5 ]
24    jal pop_front  #content is [ 2 3 4 5 ]
25    jal display
26    li $v0, 10
```

```
-- program is finished runni

ar
2,3,4,5,
-- program is finished runni
```

here we push_back 5 elements and then popfront 1 element and this is the output

---

```
13  main:
14    li $a0, 5
15    jal push_front
16    li $a0, 4
17    jal push_front
18    li $a0, 6
19    jal push_back
20    jal display
21
22
23    li $v0, 10
24    syscall
25
```

```
4,5,6,
-- program is finished runn
```

here we compine between push_front + push_back and this is the output

pop_front

C++ code

```cpp
void pop_front(){
    if (used == 0) {
        cout << "Deque is Empty :( \n" ;
        return;
    }
    front = front + 1 ;
    used -- ;
}
```

Mips code

```
67  pop_front:
68      lw $t1, frontIdx
69      lw $t2, used
70      beq $t2, $zero, Empty_error
71      addi $t1, $t1, -1
72      sw $t1, frontIdx
73      addi $t2, $t2, -1
74      sw $t2, used
75
76      jr $ra
```

Load registers:

$t1: Load the current value of frontIdx.

$t2: Load the current number of used elements (used).

Check if deque is empty:

Compare the number of used elements ($t2) with zero. If it's zero, jump to Empty_error to handle the error.

Update front index:

Increment the front index ($t1) to remove the front element.

Store the updated front index back into memory (frontIdx).

Update number of used elements:

Decrement the count of used elements ($t2) to reflect the removal of the front element.

Store the updated count back into memory (used).

Return:

Return control to the calling function using jr $ra.

pop_front

pop_front: This function delete an element at the front of the deque.

It first checks if the deque is empty, then delete the element and updates the front index and the number of used elements accordingly.

```
13  main:
14      jal pop_front
15
16
17      li $v0, 10
18      syscall
19
20
```

```
Deque is Empty :(

-- program is finished runn
```

here we try to pop_front element from empty deque and this is the error message

```
13  main:
14      li $a0,1
15      jal push_front #content is [ 1 ]
16      li $a0,2
17      jal push_front #content is [ 2 1 ]
18      li $a0,3
19      jal push_front #content is [ 3 2 1]
20
21      jal pop_back #content is [ 3 2 ]
22      jal pop_back  #content is [ 3 1 ]
23      jal display
24      li $v0, 10
25      syscall
```

Clear

```
-- program is finished running --

3,1,
-- program is finished running --
```

here we push elements then pop two elements and this is the output

pop_back

C++ code

```cpp
void pop_back(){
    if (used == 0) {
        cout << "Deque is Empty :( \n" ;
        return;
    }
    back = back - 1 ;
    used -- ;
}
```

Mips code

```
78  pop_back:
79
80      lw $t1, rearIdx
81      lw $t2, used
82      beq $t2, $zero, Empty_error
83      addi $t1, $t1, 1
84      sw $t1, rearIdx
85      addi $t2, $t2, -1
86      sw $t2, used
87
88      jr $ra
```

Load registers:

$t1: Load the current value of rearIdx.

$t2: Load the current number of used elements (used).

Check if deque is empty:

Compare the number of used elements ($t2) with zero. If it's zero, jump to Empty_error to handle the error.

Update rear index:

Increment the rear index ($t1) to remove the rear element.

Store the updated rear index back into memory (rearIdx).

Update number of used elements:

Decrement the count of used elements ($t2) to reflect the removal of the rear element.

Store the updated count back into memory (used).

Return:

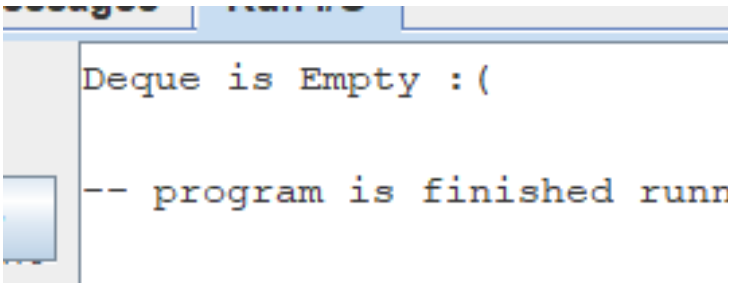
Return control to the calling function using jr $ra.

pop_back

pop_back: This function delete an element at the back of the deque.

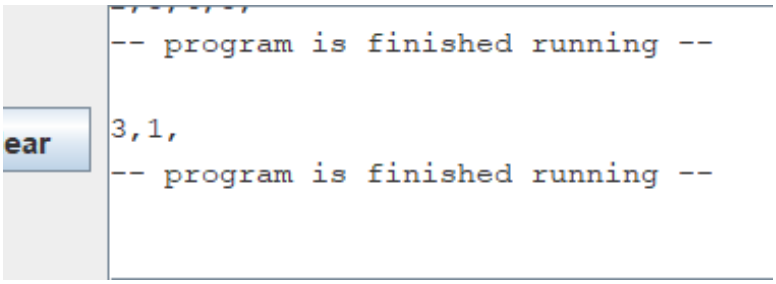
It first checks if the deque is empty, then delete the element and updates the rear index and the number of used elements accordingly.

```
13  main:
14      jal pop_back
15
16      li $v0, 10
17      syscall
18
19
```



here we try to pop_back an element from empty deque and this is the error message

```
13  main:
14      li $a0,1
15      jal push_front #content is [ 1 ]
16      li $a0,2
17      jal push_front #content is [ 2 1 ]
18      li $a0,3
19      jal push_front #content is [ 3 2 1]
20
21      jal pop_back #content is [ 3 2 ]
22      jal pop_back  #content is [ 3 1 ]
23      jal display
24      li $v0, 10
25      syscall
```



here we push elements then pop_back two elements and this is the output

display

C++ code

```cpp
void display(){
        cout << "Elements : {" ;
        for (int i = front - 1; ~i ; --i) {
            cout << deque[i] << ' ' ;
        }
        for (int i = SIZE - 1; i > back; --i) {
            cout << deque[i] << ' ' ;
        }
        cout << "}\n" ;
}
```

Mips code

deque_mips.asm*

```
91  display:
92    lw $t1, frontIdx
93    lw $t2, rearIdx
94    la $t4 ,  deque
95    displayfrontloop:
96      bltz $t1 , exitedisplayfront
97      sll $t5 , $t1 , 2
98      add $t5 , $t4 , $t5
99      lw $a0 , 0($t5)
100      li $v0, 1
101      syscall
102
103      li $v0, 4
104      la $a0, space
105      syscall
106      addi $t1  , $t1, -1
107     j displayfrontloop
108
109
110   exitedisplayfront:
111   # print the back
112   lw $t1 , MAX_SIZE
113     addi $t1  , $t1, -1
114   displayback:
115     blt $t1 , $t2 , exitedisplayback
116     sll $t5 , $t1 , 2
117     add $t5 , $t4 , $t5
118     lw $a0 , 0($t5)
119      li $v0, 1
120      syscall
121
122      li $v0, 11
123      li $a0, ' '
124      syscall
125     addi $t1  , $t1, -1
126      j displayback
127
128   exitedisplayback:
129
130     li $v0 , 4
131     la $a0 , newline
132     syscall
133     jr $ra
```

Load Indices:

lw $t1, frontIdx: Load the front index of the deque into register $t1.

lw $t2, rearIdx: Load the rear index of the deque into register $t2.

la $t4, deque: Load the base address of the deque into register $t4.

Display Front Loop:

bltz $t1, exitedisplayfront: Branch to exitedisplayfront if the front index is less than zero (indicating an empty deque).

sll $t5, $t1, 2: Shift left logical $t1 by 2 to multiply it by 4 (each element in the deque is 4 bytes), storing the result in $t5.

add $t5, $t4, $t5: Add the offset $t5 to the base address of the deque to get the address of the current element.

lw $a0, 0($t5): Load the content of the current deque element into register $a0.

li $v0, 1: Load system call code for printing an integer/string into register $v0.

syscall: Execute the system call to print the content of the deque element.

li $v0, 4: Load system call code for printing a string into register $v0.

la $a0, space: Load the address of the space character into register $a0.

syscall: Execute the system call to print a space character.

addi $t1, $t1, -1: Decrement the front index by 1.

j displayfrontloop: Jump back to displayfrontloop to continue displaying the elements.

Exit Display Front Loop:

This label marks the end of displaying elements from the front of the deque.

Display Back Loop:

blt $t1, $t2, exitedisplayback: Branch to exitedisplayback if the front index is less than the rear index (indicating all elements have been displayed).

Similar to the front loop, this loop displays elements from the rear of the deque.

Instead of printing a space, it prints a comma after each element.

Decrements the index and continues until the rear index is reached.

Exit Display Back Loop:

This label marks the end of displaying elements from the back of the deque.

Print Newline:

li $v0, 4: Load system call code for printing a string into register $v0.

la $a0, newline: Load the address of the newline character into register $a0.

syscall: Execute the system call to print a newline character.

Return:

jr $ra: Jump back to the calling subroutine.

Task assignment

Each member participated in the project with: -

	Searching and writing code
Essam Eldin Hisham	20%
Mohamed Essam Ali	16%
Abdelrhman Emad Ibrahim	16%
Abdelrahman tarek ibrahem	16%
Mohamed Ahmed Amin	16%
Mostafa Salah refaey	16%
	Reviewing code
Mohamed Essam Ali	20%
Abdelrhman Emad Ibrahim	20%
Abdelrahman tarek ibrahem	20%
Mohamed Ahmed Amin	20%
Mostafa Salah refaey	20%
	Code compilation
Essam Eldin Hisham	50%
Abdelrhman Emad Ibrahim	50%
	PowerPoint
Essam Eldin Hisham	50%
Mohamed Essam Ali	50%
	Reviewing PowerPoint
Essam Eldin Hisham	10%
Mohamed Essam Ali	10%
Abdelrhman Emad Ibrahim	10%
Abdelrahman tarek ibrahem	10%
Mohamed Ahmed Amin	10%
Mostafa Salah refaey	50%
	Report
Essam Eldin Hisham	100%

References

1 – Geeks for Geeks [link]

2 – Java point [link]

3 –Dr / Mostafa saad [udemy course]