Daria Pacheco

22 September 2022

COM 301

Homework 2

1. Due to the fact that ArrayList uses an array internally, manipulation with it is sluggish. All the remaining elements in the array are shifted in memory if any element is removed from the array, while LinkedList employs a doubly linked list, which makes it faster than ArrayList. Additionally, LinkedList excels in data manipulation, while ArrayList excels at data storage and retrieval. A default capacity of 10 is often set to an array list when it is initialized. A LinkedList never experiences default capacity. When a LinkedList is initialized, an empty list is produced in the LinkedList.

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 5 | 3 | 7 |

Execute the operation using the top two operands.

7+3=10

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 5 | 10 | 4 |

\*

Operations on the final two operands

|  |  |  |
| --- | --- | --- |
| 9 | 5 | 40 |

+

40+5

|  |  |  |
| --- | --- | --- |
| 9 | 45 | 2 |

-

45-2 = 43

|  |  |
| --- | --- |
| 9 | 43 |

+

43+9

|  |
| --- |
| 387 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Stack** | **Output** | **Operation** |
| A+B\*C+(D\*E+F) | Empty | - | Initial |
| +B\*C+(D\*E+F) | Empty | A | Print |
| B\*C+(D\*E+F) | +1 | A | Push |
| \*C+(D\*E+F) | +1 | AB | Print |
| C+(D\*E+F) | +1\* | AB | Push |
| +(D\*E+F) | +1\* | ABC | Print |
| (D\*E+F) | +1 | ABC\*+ | Pop twice then push |
| D\*E+F) | +1(1 | ABC\*+ | Push |
| \*E+F) | +1(1 | ABC\*+D | Print |
| E+F) | +1(1\* | ABC\*+D | Push |
| +F) | +1(1\*1 | ABC\*+DE | Print |
| F) | +1(1+ | ABC\*+DE\* | Pop twice then push |
| ) | +1(1+ | ABC\*+DE\*F | Push |
| - | + | ABC\*+DE\*F+ | Pop till ( |
| Empty | Empty | ABC\*+DE\*F++ | Pop |

Postfix Expression = ABC\*+DE\*F++

1. #include <iostream>

#define SIZE 5

using namespace std;

class Queue {

private:

int items[SIZE], front, rear;

public:

Queue() {

front = -1;

rear = -1;

}

bool isFull() {

if (front == 0 && rear == SIZE - 1) {

return true;

}

return false;

}

bool isEmpty() {

if (front == -1)

return true;

else

return false;

}

void enQueue(int element) {

if (isFull()) {

cout << "Queue is full";

} else {

if (front == -1) front = 0;

rear++;

items[rear] = element;

cout << endl

<< "Inserted " << element << endl;

}

}

int deQueue() {

int element;

if (isEmpty()) {

cout << "Queue is empty" << endl;

return (-1);

} else {

element = items[front];

if (front >= rear) {

front = -1;

rear = -1;

} /\* Q has only one element, so we reset the queue after deleting it. \*/

else {

front++;

}

cout << endl

<< "Deleted -> " << element << endl;

return (element);

}

}

void display() {

/\* Function to display elements of Queue \*/

int i;

if (isEmpty()) {

cout << endl

<< "Empty Queue" << endl;

} else {

cout << endl

<< "Front index-> " << front;

cout << endl

<< "Items -> ";

for (i = front; i <= rear; i++)

cout << items[i] << " ";

cout << endl

<< "Rear index-> " << rear << endl;

}

}

};

int main() {

Queue q;

//deQueue is not possible on empty queue

q.deQueue();

q.enQueue(1);

q.enQueue(2);

q.enQueue(3);

q.enQueue(4);

q.enQueue(5);

// 6th element can't be added to because the queue is full

q.enQueue(6);

q.display();

q.deQueue();

q.display();

return 0;

}

Graphical user interface, text, application, Word

Description automatically generated

Table

Description automatically generated