University American College Skopje
SINCE 2005

DATA STRUCTURES AND ALGORITHMS

- LECTURE 7-

(2ND YEAR OF STUDY)

Skopje, 2023/24

Contents

7. Stacks and Queues

7.1. Stacks

7.2. Queues

7.1. Stacks

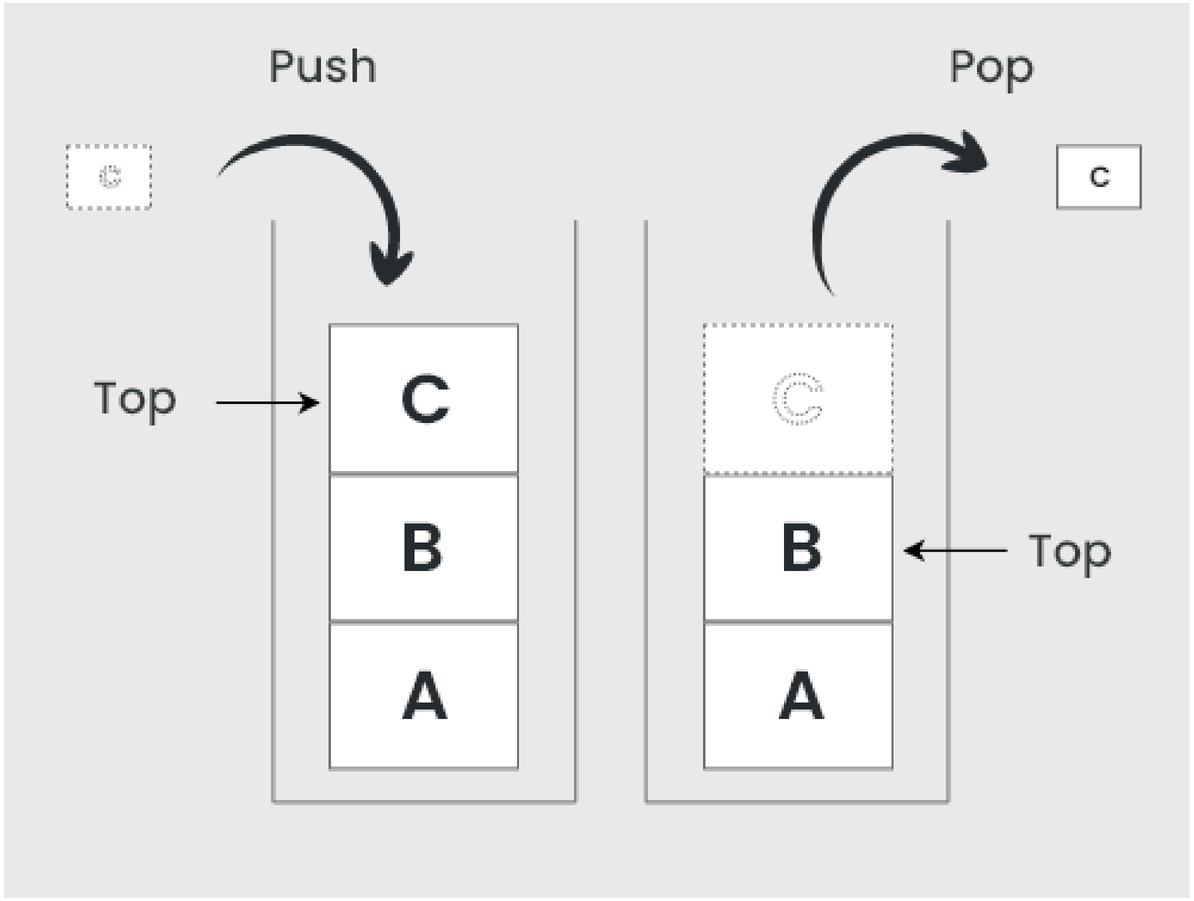
- **Stack** is a linear data structure that follows a particular order in which the operations are performed.
- The order may be LIFO (Last In First Out) or FILO (First In Last Out).
- LIFO implies that the element that is inserted last comes out first, and FILO implies that the element that is inserted first comes out last. See the below figure:

7.1. Stacks

- A **Stack** is a linear data structure in which the *insertion* of a new element, and *removal* of an existing element takes place at the same end represented as the ***top*** of the stack.
- To implement the stack, it is required to maintain the ***pointer to the top of the stack***, which is the last element to be inserted because ***we can access the elements only on the top of the stack.***

7.1. Stacks

- **Stack** data structure:



7.1. Stacks

- Stacks are a type of *container adaptors* with FILO/LIFO type of working, where a new element is added at one end (top), and an element is removed from that end only.
- Stack uses an encapsulated object of either **vector** or **deque** (by default), or **list** (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

7.1. Stacks

- For creating a stack, we must include the <stack> header file in our code.
- The functions associated with stack are:
  - **empty()** – Returns whether the stack is empty – Time Complexity : O(1)
  - **size()** – Returns the size of the stack – Time Complexity : O(N)
  - **top()** – Returns a reference to the top most element of the stack – Time Complexity : O(1)
  - **push(g)** – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)
  - **pop()** – Deletes the most recent entered element of the stack – Time Complexity : O(1)

7.1. Stacks

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> stack;
    stack.push(21);// The values pushed in the stack should be of the
    stack.push(22);   same data which is written during declaration of stack
    stack.push(24);
    stack.push(25);
    int num=0;
      stack.push(num);
    stack.pop();
    stack.pop();
      stack.pop();

    while (!stack.empty()) {
        cout << stack.top() <<" ";
        stack.pop();
    }
}
```

Output

```
22 21
```

7.1. Stacks

- **Time complexity:** The time complexity of this program is ***O(N)***, where ***N*** is the total number of elements in the stack. The while loop iterates ***N*** times, popping elements from the stack and printing them.
- **Space complexity:** The space complexity of this program is ***O(N)***, where ***N*** is the total number of elements in the stack. The stack data structure uses space proportional to the number of elements stored in it. In this case, the maximum size of the stack is 5, so the space complexity is constant and can be considered as O(1) as well.

7.1. Stacks

```cpp
// CPP program to illustrate
// Implementation of size() function
#include <iostream>
#include <stack>
using namespace std;

int main()
{
    int sum = 0;
    stack<int> mystack;
    mystack.push(1);
    mystack.push(8);
    mystack.push(3);
    mystack.push(6);
    mystack.push(2);

    // Stack becomes 1, 8, 3, 6, 2

    cout << mystack.size();

    return 0;
}
```

Output:

```
5
```

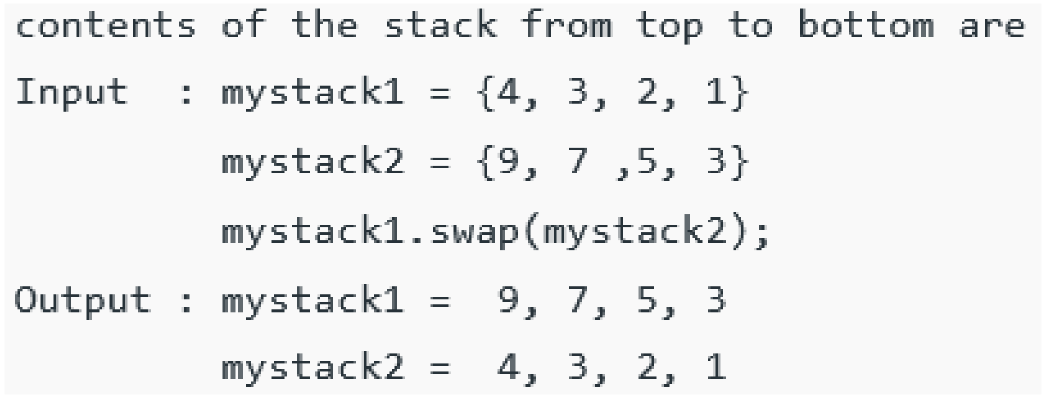
```cpp
// CPP program to illustrate
// Application of size() function
#include <iostream>
#include <stack>
using namespace std;

int main()
{
    int sum = 0;
    stack<int> mystack;
    mystack.push(1);
    mystack.push(8);
    mystack.push(3);
    mystack.push(6);
    mystack.push(2);

    // Stack becomes 1, 8, 3, 6, 2

    while (mystack.size() > 0) {
        sum = sum + mystack.top();
        mystack.pop();
    }
    cout << sum;
    return 0;
}
```

Output:

```
20
```

7.1. Stacks

- The function **swap()** is used to swap the contents of one stack with another stack of same type, but the size may vary.
- **Syntax :** *stackname1*.swap(*stackname2*)
    - **Parameters:** The name of the stack with which the contents have to be swapped.
    - **Result:** All the elements of the 2 stack are swapped.



```
contents of the stack from top to bottom are
Input  : mystack1 = {4, 3, 2, 1}
         mystack2 = {9, 7 ,5, 3}
         mystack1.swap(mystack2);
Output : mystack1 =  9, 7, 5, 3
         mystack2 =  4, 3, 2, 1
```

7.1. Stacks

- **Applications of Stack Data Structure:**
  - **Function calls and recursion:** When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution.
  - **Undo/Redo operations:** The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.
  - **Expression evaluation:** Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements.

7.1. Stacks

- **Browser history:** Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.
- **Balanced Parentheses:** Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced.
- **Backtracking Algorithms:** The backtracking algorithm uses stack to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack.

7.1. Stacks

- **Applications of Stack in real life:**
  - CD/DVD stand.
  - Stack of books in a book shop.
  - Call center systems.
  - Undo and Redo mechanism in text editors.
  - The history of a web browser is stored in the form of a stack.
  - Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
  - YouTube downloads and Notifications are also shown in LIFO format (the latest appears first).
  - Allocation of memory by an operating system while executing a process.

7.2. Queues

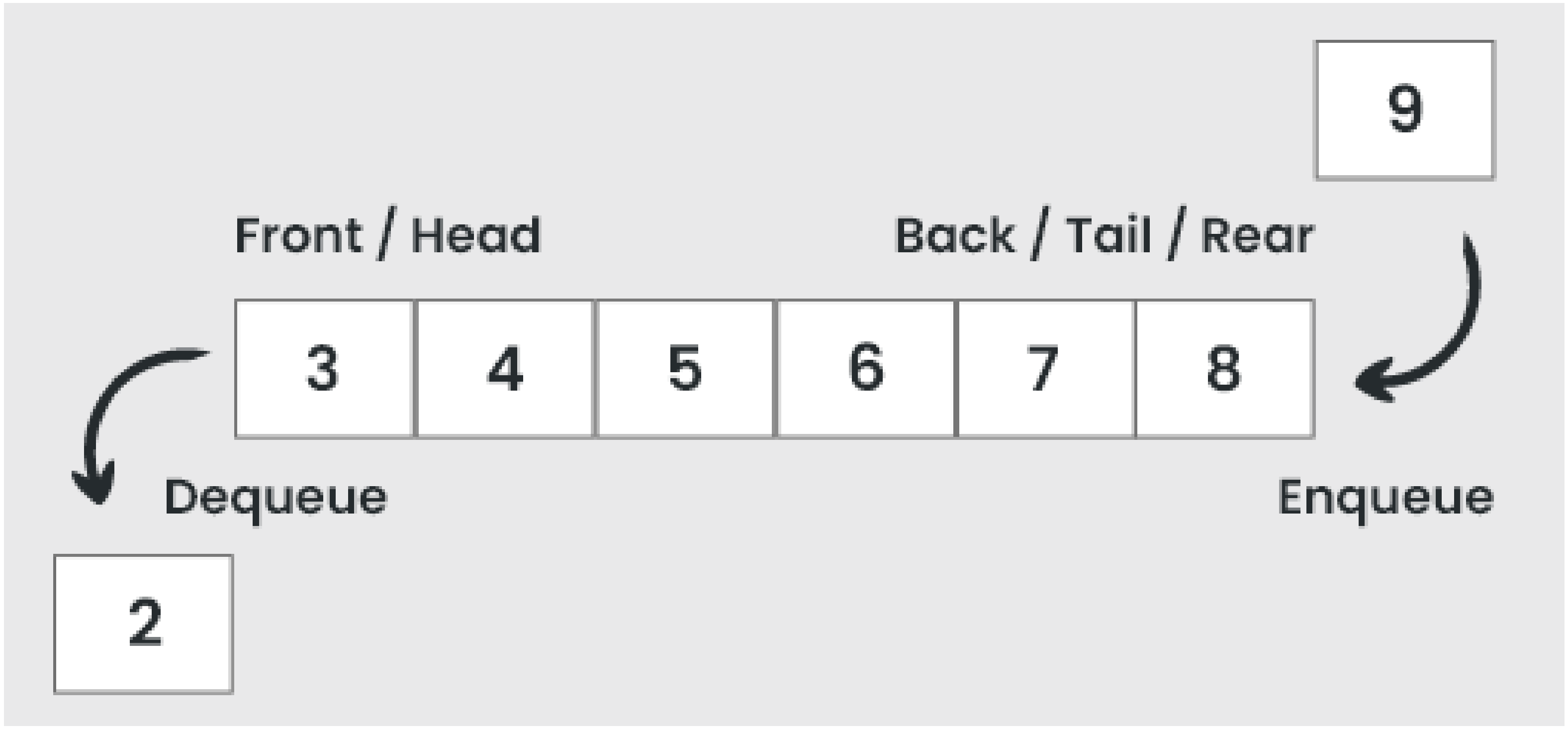
- A **Queue** is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.
- We define a Queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end.
- The element which is first pushed into the order, the operation is first performed on that.

7.2. Queues

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **Front** of the queue (sometimes, **Head** of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **Rear** (or, the **Tail**) of the queue.

7.2. Queues

- **Queue** data structure:



7.2. Queues

- **Types of Queue:**
  - **Input Restricted Queue:** In this queue, the input can be taken from only one end, but deletion can be done from any of the ends.
  - **Output Restricted Queue:** In this queue, the input can be taken from both ends, but deletion can be done from only one end.
  - **Circular Queue:** This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order.
  - **Double-Ended Queue (Dequeue):** In this queue the insertion and deletion operations, both can be performed from both ends.
  - **Priority Queue:** A priority queue is a special queue where the elements are accessed based on the priority assigned to them.

7.2. Queues

- Queues are a type of **container adaptors** that operate in a First In First Out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.
- Queues use an encapsulated object of **deque** or **list** (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.
- For creating a queue, we must include the <queue> header file in our code.

7.2. Queues

- The functions associated with queues are:
  - **empty()** – Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.
  - **size()** – Returns the size of the queue.
  - **front()** – Returns a reference to the first element of the queue.
  - **back()** – Returns a reference to the last element of the queue.
  - **push(g)** – Adds the element 'g' at the end of the queue.
  - **pop()** – Deletes the first element of the queue.
  - **swap()** – Exchange the contents of two queues, but the queues must be of the same data type, although sizes may differ.
- The time complexity for all operations: O(1), but for the operation *size()*: O(N).

7.2. Queues

```
#include <iostream>
#include <queue>

using namespace std;

// Print the queue
void print_queue(queue<int> q)
{
    queue<int> temp = q;
    while (!temp.empty()) {
        cout << temp.front()<<" ";
        temp.pop();
    }
    cout << '\n';
}

// Driver Code
int main()
{
    queue<int> q1;
    q1.push(1);
    q1.push(2);
    q1.push(3);

    cout << "The first queue is : ";
    print_queue(q1);
```

```
 queue<int> q2;
q2.push(4);
q2.push(5);
q2.push(6);

cout << "The second queue is : ";
print_queue(q2);


  q1.swap(q2);

  cout << "After swapping, the first queue is : ";
print_queue(q1);
  cout << "After swapping the second queue is : ";
print_queue(q2);

  cout<<q1.empty();  //returns false since q1 is not empty

return 0;
}
```

Output

```
The first queue is : 1 2 3
The second queue is : 4 5 6
After swapping, the first queue is : 4 5 6
After swapping the second queue is : 1 2 3
0
```

7.2. Queues

- **Time complexity:** The time complexity of this program is ***O(N)***, where ***N*** is the number of elements in the queue. The *print_queue(q)* function has time complexity of O(*N*), while the *q1.push(1)-q2.push(6)* functions have time complexity of O(*1*) for each push operation.
- **Space complexity:** The space complexity of this program is ***O(N)***, where ***N*** is the number of elements in the queue. The *print_queue(q)* function has space complexity of O(*N*), while the *q1.push(1)-q2.push(6)* functions have space complexity of O(*N*).

7.2. Queues

- By using *std::dequeue*, we can iterate with efficient space which provides all the standard operations of *std::queue*.
- We can use:
  - *dequeue::cbegin()* and *dequeue::cend()* to print dequeue in forward direction, and
  - *dequeue::crbegin()* and *dequeue::crend()* to print dequeue in backward direction.

7.2. Queues

```cpp
// C++ program to iterate a STL Queue
// using std : : dequeue
#include <iostream>
#include <queue>
#include <algorithm>
using namespace std;
int main()
{
    deque<int> dq;
    dq.push_back(1);
    dq.push_back(2);
    dq.push_back(3);
    dq.push_back(4);
    dq.push_back(5);

    cout << "Printing dequeue in forward direction : \n";
    for(auto it = dq.cbegin();it!=dq.cend();it++)
    {
        cout << *it << " ";
    }

    cout << "\n";
    cout << "Printing dequeue in reverse direction : \n";
    for(auto it = dq.crbegin();it!=dq.crend();it++)
    {
        cout << *it << " ";
    }

    return 0;
}
```

Output

```
Printing dequeue in forward direction :
1 2 3 4 5
Printing dequeue in reverse direction :
5 4 3 2 1
```

Time complexity: O(n) // n is the size of the queue.

Auxiliary space: O(n).

7.2. Queues

- **Applications of Queue Data Structure:**
  - **Multi programming:** It means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these ones are organized as queues.
  - **Network:** In a network, a queue is used in devices such as: a router, or a switch. Another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.
  - **Job Scheduling:** The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one, which is organized using a queue.
  - **Shared resources:** Queues are used as waiting lists for a single shared resource.

7.2. Queues

- **Real-time applications of Queue:**
  - ATM Booth Line
  - Ticket Counter Line
  - Key press sequence on the keyboard
  - CPU task scheduling
  - Waiting time of each customer at call centers