

# CS213/293 Data Structure and Algorithms 2025

## Lecture 3: Stack and queue

Instructor: Ashutosh Gupta

IITB India

Compile date: 2025-08-09

# Topic 3.1

## Stack

# Stack

## Definition 3.1

A **stack** is a container where elements are added and deleted according to the last-in,first-out (LIFO) order.

- ▶ Addition is called **pushing**
- ▶ Deleting is called **popping**

## Example 3.1

- ▶ A stack of papers in a copier
- ▶ Undo-redo features in editors
- ▶ Back button on the Browser

# Interface of stack

Reference: <https://en.cppreference.com/w/cpp/container/stack>

Stack supports four interface methods.

- ▶ `stack<T> s` : Allocates a new stack `s`.
- ▶ `s.push(e)` : Pushes the given element `e` to the top of the stack.
- ▶ `s.pop()` : Removes the top element from the stack.
- ▶ `s.top()` : Accesses the top element of the stack.

Some support functions

- ▶ `s.empty()` : checks whether the stack is empty
- ▶ `s.size()` : returns the number of elements

## Exercise 3.1

Why do we define stack when we can use vector for the same effect?

**Commentary:** Answer: vector in C++ promises to provide efficient random access, but stack does not make such a promise. Therefore, the implementations of the stack may make implementation choices that may result in inefficient random access.

## Axioms of stack\*

Let  $s1$  and  $s$  be stacks.

- ▶ `Assume(s1 == s); s.push(e); s.pop(); Assert(s1==s);`
- ▶ `s.push(e); Assert(s.top()==e);`

`Assume(s1 == s)` means that we **assume** that the contents of  $s1$  and  $s$  are the same.

`Assert(s1 == s)` means that we **check** that the contents of  $s1$  and  $s$  are the same.

# Exercise: action on the empty stack

## Exercise 3.2

Let `s` be an empty stack in C++.

- ▶ What happens when we run `s.top()` ?
- ▶ What happens when we run `s.pop()` ?

Ask ChatGPT.

**Commentary:** Answer: `s.top()` will cause a segmentation fault. `s.pop()` will not cause any error and exit without any effect. The behavior depends on the compiler. Compare the behavior of the program under g++ 13.3.0 and 15.1.1.

## Topic 3.2

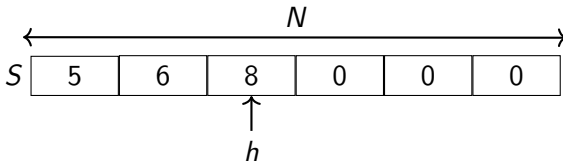
### Implementing stack

## Array-based stack

Let us look at a simplified array-based implementation of an stack of integers.

The stack consists of three variables.

- ▶  $N$  specifies the currently available space in the stack.
- ▶  $S$  is the integer array of size  $N$
- ▶  $h$  is the position of the head of the stack.





## Implementing stack

```
class arrayStack {
    int    N = 2;        // Capacity
    int*   S = NULL;     // pointer to array
    int    h = -1;       // Current head of the stack
public:
    arrayStack() { S = (int*)malloc(sizeof(int)*N ); }
    int    size()  { return h+1;  }
    bool   empty() { return h<0;  }
    int    top()   { return S[h]; } // On empty stack what happens?
    void   push(int e) {
        if( size() == N ) expand(); // Expand capacity of the stack
        S[++h] = e;
    }
    void   pop() { if( !empty() ) h--; }
```

**Commentary:** The behavior of the above implementation may not match the behavior of the C++ stack library. To ensure a segmentation fault in top(), when the stack is empty one may use the following code. `if( empty() ) return *(int*)0; else return S[t];`

## Implementing stack (expanding when full)

```
private:
    void expand() {
        int new_size = N*2; // We observed the growth in our lab!!
        int* tmp = (int*) malloc( sizeof(int)*new_size ); //New array
        for( unsigned i =0; i < N; i++ ) { // copy from the old array
            tmp[i] = S[i];
        }
        free(S);           // Release old memory
        S = tmp;           // Update local fields
        N = new_size; //
    }
};
```

### Exercise 3.3

Can we utilize the DMA (Direct Memory Access) feature of processors to perform the copy?

# Efficiency

All operations are performed in  $O(1)$  if there is no expansion to the stack capacity.

What is the cost of expansion?

## Topic 3.3

Why exponential growth strategy?

# Growth strategy

Let us consider two possible choices for growth.

- ▶ Constant growth:  $\text{new\_size} = N + c$  (for some fixed constant  $c$ )
- ▶ Exponential growth:  $\text{new\_size} = 2*N$

Which of the above two is better?

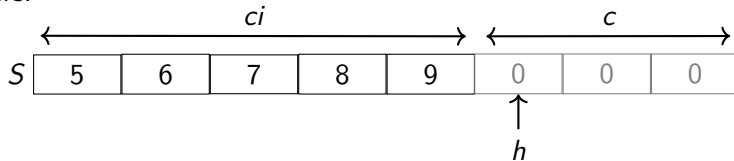
## Analysis of constant growth

Let us suppose initially  $N = 0$  and there are  $n$  consecutive pushes.

After every  $c$ th push, there will be an expansion operation.

Therefore, the expansion operation at  $(ci + 1)$ th push will

- ▶ allocate memory of size  $c(i + 1)$  and
- ▶ copy  $ci$  integers.



Cost of  $i$ th expansion:  $c(2i + 1)$ .

**Commentary:** We are assuming that allocating memory of size  $k$  costs  $k$  time, which may be more efficient in practice. Bulk memory copy can also be sped up by vector instructions.

## Analysis of constant growth(2)

For  $n$  pushes, there will be  $n/c$  expansions.

The total cost of expansions:

$$c(1 + 3 + \dots + (2\frac{n}{c} + 1)) = c(n/c)^2 \in O(n^2)$$

Nonlinear cost!

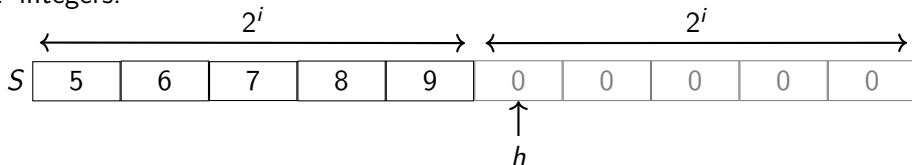
## Analysis of exponential growth

Let us suppose initially  $N = 1$  and there are  $n = 2^r$  consecutive pushes.

The expansion operations will only occur at the  $(2^i + 1)$ th push, where  $i \in [0, r - 1]$ .

The expansion operation at  $2^i + 1$ th push will

- ▶ allocate memory of size  $2^{i+1}$  and
- ▶ copy  $2^i$  integers.



Cost of the expansion:  $3 * 2^i$ .



## Analysis of exponential growth(2)

For  $2^r$  pushes, the last expansion would be at  $2^{r-1} + 1$ .

The total cost of expansions:

$$3(2^0 + \dots + 2^{r-1}) = 3 * (2^r - 1) = 3 * (n - 1)$$

Linear cost! The average cost of push remains  $O(1)$ .

### Exercise 3.4

Why double? Why not triple? Why not 1.5 times? Is there a trade-off?

**Commentary:** The policy is not the same in all the libraries for vectors. What is the policy in the following implementations of STL?  
GCC STL: (libstdc++) [https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl\\_vector.h#L2187](https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_vector.h#L2187)  
Clang STL: (libc++) [https://github.com/llvm/llvm-project/blob/main/libcxx/include/\\_\\_vector/vector.h#L909](https://github.com/llvm/llvm-project/blob/main/libcxx/include/__vector/vector.h#L909)  
MASL STL: <https://github.com/microsoft/STL/blob/main/stl/inc/vector#L2006>

## Topic 3.4

### Applications of stack

# Stacks are everywhere

Stack is a foundational data structure.

It shows up in a vast range of algorithms.

## Example: matching parentheses

Problem:

Given an input text, check if it has matching parentheses.

Examples:

▶ "{a[sic]tik}" ✓

▶ "{a[sic}tik}" ✗

```
bool parenMatch(string text ) {  
    std::stack<char> s;  
    for(char c : text ) {  
        if( c == '{' or c == '[' ) s.push(c);  
        if( c == '}' or c == ']' ) {  
            if( s.empty() ) return false;  
            if( c-s.top() != 2 ) return false;  
            s.pop();  
        }  
    }  
    if( s.empty() ) return true;  
    return false;  
}
```

## Topic 3.5

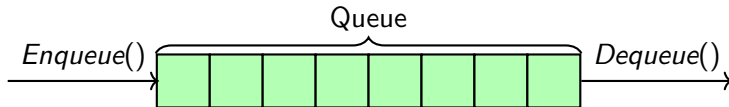
### Queue

# Queue

## Definition 3.2

**Queue** is a container where elements are added and deleted according to the first-in-first-out (FIFO) order.

- ▶ Addition is called **enqueue**
- ▶ Deleting is called **dequeue**



## Example 3.2

- ▶ Entry into an airport
- ▶ Calling a lift in a building (slightly different; it is a priority queue)

# Interface of queue

Reference: <https://en.cppreference.com/w/cpp/container/queue>

Queue supports four main interface methods

- ▶ `queue<T> q` : allocates a new queue `q`.
- ▶ `q.enqueue(e)` : Adds the given element `e` to the end of the queue. (push)
- ▶ `q.dequeue()` : Removes the first element from the queue. (pop)
- ▶ `q.front()` : access the first element .

Some support functions

- ▶ `q.empty()` : checks whether the queue is empty.
- ▶ `q.size()` : returns the number of elements.

**Commentary:** All literature uses the terms enqueue and dequeue, but unfortunately, the C++ library uses push for enqueue and uses pop for dequeue. Other languages, such as Java, use the terms enqueue and dequeue.

## Axioms of queue\*

1. `queue<T> q; Assert(q.empty() == true);`
2. `q.enqueue(e); Assert(q.empty() == false);`
3. `Assume(q.empty() == true);`  
`q.enqueue(e); Assert(q.front() == e);`
4. `Assume(q.empty() == false && old_q == q);`  
`q.enqueue(e); Assert(old_q.front() == q.front());`
5. `Assume(q.empty() == true && old_q == q);`  
`q.enqueue(e); q.dequeue(); Assert(old_q == q);`
6. `Assume(q.empty() == false && q == q1);`  
`q.enqueue(e); q.dequeue(); q1.dequeue(); q1.enqueue(e); Assert(q == q1);`

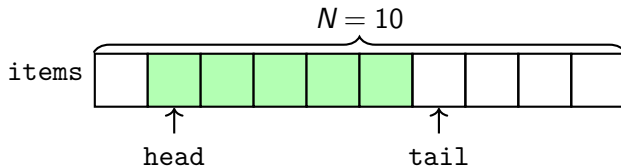


## Topic 3.6

### Array implementation of queue

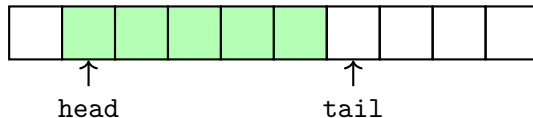
# Array-based implementation

- ▶ Queue is stored in an array `items` in a circular fashion
- ▶ Three integers record the state of the queue
  1. `N` indicates the available capacity ( $N-1$ ) of the queue
  2. `head` indicates the position of the front of the queue
  3. `tail` indicates position one after the rear of the queue

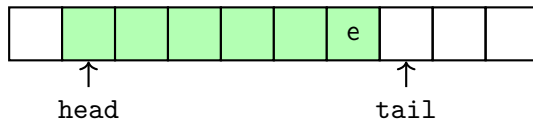


## Enqueue operation on the array

Consider the following state of the queue.

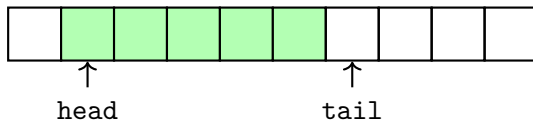


After the enqueue(e) operation:

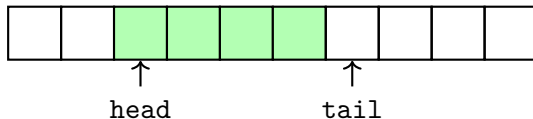


## Deque operation on the array

Consider the following state of the queue.



After the dequeue() operation:

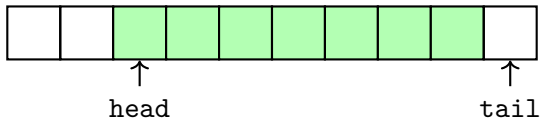


### Exercise 3.5

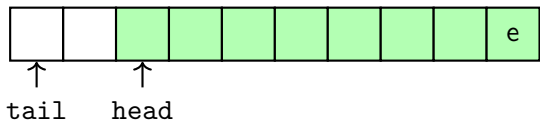
1. Where will `front()` read from?
2. What is the size of the queue?

## Wrap around to utilize most of the array

Consider the following state of the queue.

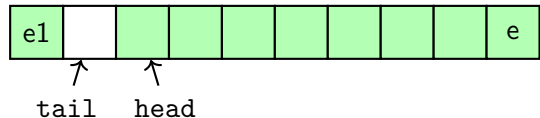


After the enqueue(e) operation, we move the tail to 0.



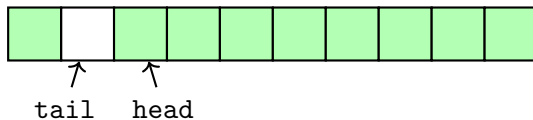
Wrap-around allows us to use the array repeatedly.

After another enqueue(e1) operation:

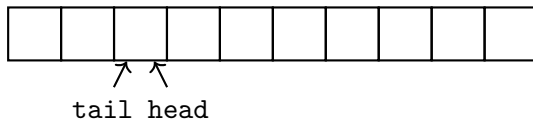


## Full and empty queue

Full queue:



Empty queue:



### Exercise 3.6

Can we use all  $N$  cells for storing elements?

## Array implementation

The code is not written in C++; We will slowly move towards pseudo code to avoid clutter on slides.

```
int head = 0, tail=0, N = INITIAL_CAPACITY;
```

```
Object items[N];           // Some initial size
```

```
bool empty()    { return (head == tail); }
```

```
int size()      { return (N+tail-head)%N; }
```

```
Object front() { return items[head]; }
```

## Array implementation(2)

```
void dequeue() {  
    if( empty() ) throw Empty;           //Queue is empty  
    free(items[head]); items[head] = NULL; //Clear memory  
    head = (head+1)%N;                   //Remove an element  
}  
  
void enqueue( Object x ) {  
    if ( size() == N-1 ) expand(); //expand when queue is full  
    items[tail] = x;  
    tail = (tail+1)%N; //insert element  
}
```

### Exercise 3.7

In our stack implementation, we did not invoke free in pop, but we invoked free in dequeue.

Why?

**Commentary:** In the stack implementation, we were only handling a stack of int. Here, we are handling a queue of **arbitrary objects**. If the object has a dynamic size, then it cannot live on the queue itself. We will store a reference to the object on the queue and allocate the object somewhere else. Therefore, we must free the objects on dequeue (the above syntax of free is not in C++; It will only work when items[head] is a reference). However, there is no need to free an int because it has a fixed size (32 bits  $\leq$  size of pointers=64bits) and it was stored on the array of the stack itself.



## Topic 3.7

### Queue via linked lists

# Can we avoid expansion?

Arrays need expansion whenever they are full.

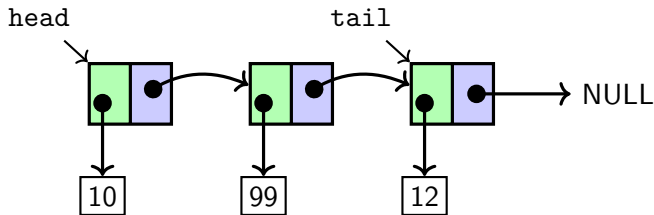
Can we expand without the need for copying?

We may use linked lists to achieve this.

# Linked lists

## Definition 3.3

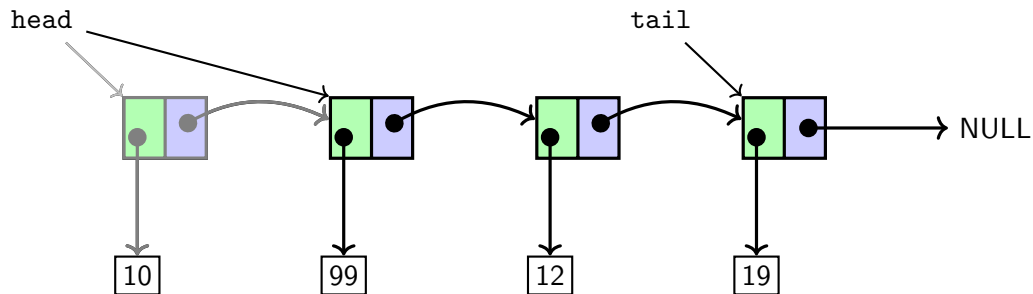
A linked list consists of nodes (small pieces of memory) with two fields: **data** and **next** pointer. The nodes form a chain via the next pointer. The data pointers point to the objects that are stored on the linked list.



## Exercise 3.8

- If we use a linked list for implementing a queue, which side should be the front of the queue?
- What is the fundamental difference between an array and a linked list?

## Deque in linked lists

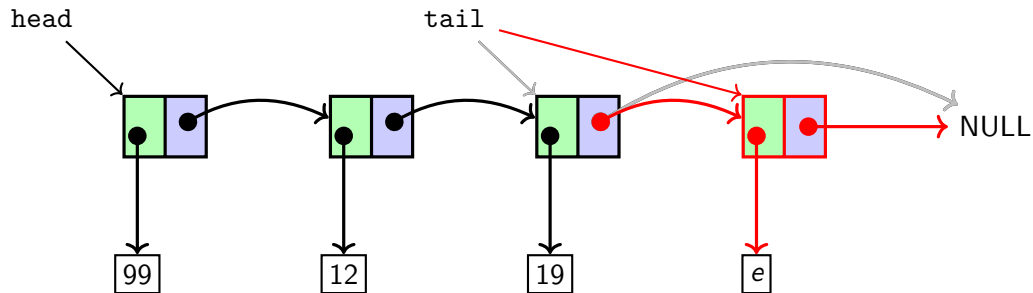


### Exercise 3.9

What happens to the object containing 10?

**Commentary:** Answer: There are several choices. The object is deallocated, the reference to the object is returned to the caller, or the copy of the object is returned to the caller. Different implementations may do it differently. This behavior is not part of the specification of the queue.

## Enqueue(e) in linked lists



### Exercise 3.10

- Which one is better: array or linked list?
- Do we need the tail pointer?

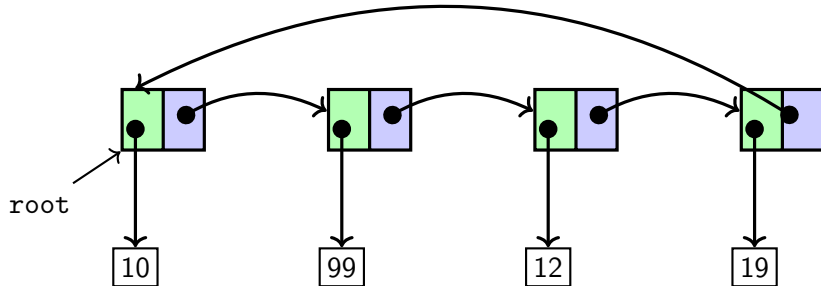
## Topic 3.8

### Circular linked list

# Circular linked list

## Definition 3.4

In a circular linked list, the nodes form a circular chain via the `next` pointer.



A root pointer points at some node of the circular list.

# Queue via circular linked lists

We may use the circular linked list for queue.

For queue, a single `root` pointer can do the job of the `head` and `tail`.

- ▶ `tail = root`
- ▶ `head = root->next`

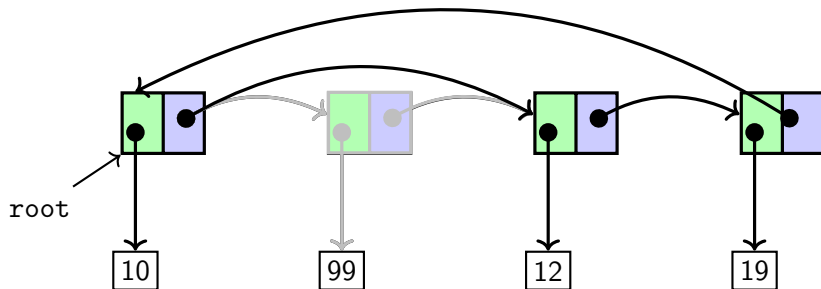
## Exercise 3.11

Which element should be returned by `front()`?

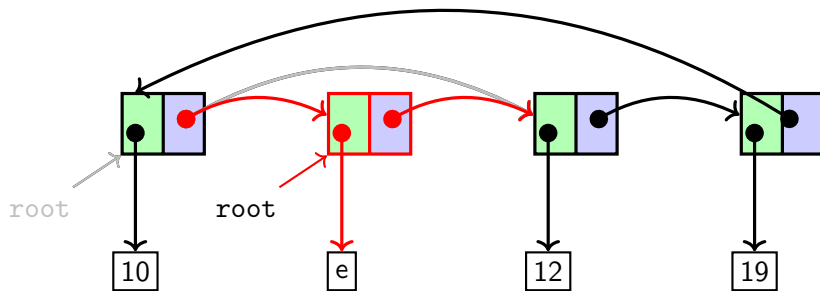
**Commentary:** Please note that we cannot implement queue if we try using `root` as `head` and `root->next` as `tail`.



## Dequeue in circular linked lists



## enqueue(e) in circular linked lists



### Exercise 3.12

Give pseudo code of the implementation of queue using a circular linked list.(Midsem 2023)

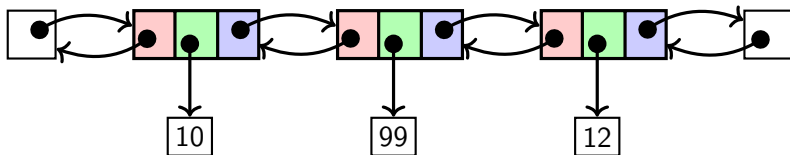
## Topic 3.9

### Doubly linked list and Deque

# Doubly linked lists

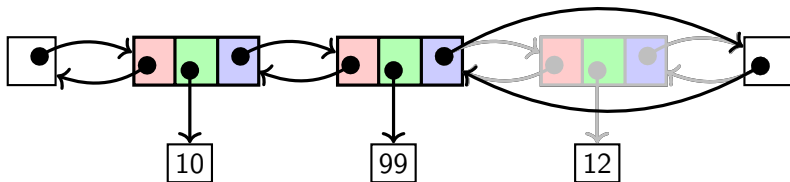
## Definition 3.5

A doubly linked list consists of nodes with three fields: **prev**, **data**, and **next** pointer. The nodes form a bidirectional chain via the **prev** and **next** pointers. The **data** pointers point to the objects that are stored on the linked list.



At both ends, two **dummy or sentinel** nodes do not store any data and are marker of termination of the list.

## Deleting a node in a doubly linked list



### Exercise 3.13

Give steps to insert a node.

# Deque (Double-ended queue)

## Definition 3.6

**Deque** is a container where elements are added and deleted according to both last-in-first-out (LIFO) and first-in-first-out (FIFO) order.

# Interface of Deque

Reference: <https://en.cppreference.com/w/cpp/container/deque>

Deque supports six main interface methods

- ▶ `deque<T> q` : allocates a new deque `q`.
- ▶ `q.push_back(e)` : Adds element `e` to the back.
- ▶ `q.push_front(e)` : Adds element `e` to the front.
- ▶ `q.pop_front()` : Removes the first element.
- ▶ `q.pop_back()` : Removes the last element.
- ▶ `q.front()` : accesses the first element.
- ▶ `q.back()` : accesses the last element.

Other support functions are similar to queue.

We can implement the Deque data structure using the doubly linked list.

# Stack and queue via Deque

We can implement both the stack and the queue using the interface of the deque.

## Exercise 3.14

- ▶ Which functions of deque implement a stack?
- ▶ Which functions of deque implement a queue?

All modification operations are implemented in  $O(1)$ .

## Exercise 3.15

Can we implement `size` in  $O(1)$  in a doubly linked list?



## Topic 3.10

### Tutorial problems

## Exercise: Use of stack

### Exercise 3.16

The span of a stock's price on the  $i$ th day is the maximum number of consecutive days (up to the  $i$ th day) the price of the stock has been less than or equal to its price on day  $i$ .

Example: for the price sequence 2 4 6 3 5 7 of a stock, the span of prices is 1 2 3 1 2 6.

Give a linear-time algorithm that computes  $s_i$  for a given price series.

## Exercise: flipping Dosa

### Exercise 3.17

There is a stack of dosas on a tava, of distinct radii. We want to serve the dosas of increasing radius. Only two operations are allowed: (i) serve the top dosa, (ii) insert a spatula (flat spoon) in the middle, say after the first  $k$ , hold up this partial stack, flip it upside-down, and put it back. Design a data structure to represent the tava, input a given tava, and produce an output in sorted order. What is the time complexity of your algorithm?

This is also related to the train-shunting problem.

## Exercise: exponential growth

### Exercise 3.18

- a. Analyze the performance of exponential growth if the growth factor is three instead of two. Does it give us better or worse performance than the doubling policy?
- b. Can we do a similar analysis for growth factor 1.5?

## Exercise: reversing a linked list (Quiz 2024)

### Exercise 3.19

Give an algorithm to reverse a linked list. You must use only three extra pointers.

## Exercise: middle element

### Exercise 3.20

Give an algorithm to find the middle element of a singly linked list.

## Exercise: stack and queue (Endsem 2023)

### Exercise 3.21

Given two stacks  $S1$  and  $S2$  (working in the LIFO method) as black boxes, with the regular methods: “Push”, “Pop”, and “isEmpty”, you need to implement a Queue (specifically : Enqueue and Dequeue working in the FIFO method). Assume there are  $n$  Enqueue/ Dequeue operations on your queue. The time complexity of a single method, Enqueue or Dequeue, may be linear in  $n$ , however the total time complexity of the  $n$  operations should also be  $\Theta(n)$ .

# Topic 3.11

## Problems



# True or False

## Exercise 3.22

Mark the following statements as True/False and also justify.

1. The element at the top of the stack has been in the stack for the longest time.
2. A stack does not allow random access to its elements.
3.  $O(1)$  is the worst-case time complexity of dequeue in a queue containing  $m$  elements implemented using an array.
4. The pop operation on a stack is always  $O(1)$ .
5. A stack can be implemented using a doubly-linked list as well as a singly-linked list
6. We can traverse a singly linked list backward, starting from the last node to the first node.
7. In C++, `std::queue<T>` has method `push_back`.
8. In C++ `std::vector<T>`, the buffer size is doubled whenever there is a call to `push_back` after the vector is full.

## Problem: messy queue

### Exercise 3.23

The mess table queue problem: There is a common mess for  $k$  hostels. Each hostel has some  $N_1, \dots, N_k$  students. These students line up to pick up their trays in the common mess. However, the queue is implemented as follows: If a student sees a person from his/her hostel, she/he joins the queue behind this person. This is the "enqueue" operation. The "dequeue" operation is, as usual, at the front. Think about how you would implement such a queue. What would be the time complexity of enqueue and dequeue? Do you think the average waiting time in this queue would be higher or lower than that of a normal queue? Would there be any difference in any statistic? If so, what?

# Merge sorted linked lists (Quiz 2023)

## Exercise 3.24

Write a time and space-efficient algorithm to merge  $k$  sorted-linked lists in sorted order, each containing the same no of elements?

## Exercise: axioms of queue\*\*

### Exercise 3.25

Using axioms of queue, show that the assert in the following does not fail.

```
queue<int> q, q1;  
q.enqueue(2);  
q.enqueue(0);  
q.enqueue(7);  
q.dequeue();  
q.dequeue();  
q.enqueue(3);
```

```
q1.enqueue(7);  
q1.enqueue(3);  
Assert(q == q1);
```

## Topic 3.12

Extra slides: Discussion

## Point of discussion: vector instructions for fast expansion\*

### Exercise 3.26

Save the following code in `copy.cpp` and compile with the following options.

```
void copy(int* dst, int* src) {  
    for( unsigned i =0; i < 32; i++ ) {  
        dst[i] = src[i];  
    }  
}
```

► Compile with: `g++ copy.cpp -O3 -S`

► Compile with: `g++ copy.cpp -O3 -mavx512f -S` (Enables vector instructions!)

Read the assembly generated in the `copy.s` file. Observe the difference.

## Point of discussion: store values or references in the collection\*

A collection like a queue has two choices: either **store values** or **references to the values**.

Operations on the collection need to behave differently. For example,

Operation	Storing values	Storing references
<code>enqueue(Object e)</code>	store a copy of the e	store a reference to e
<code>front()</code>	return a copy	return a reference
<code>dequeue()</code>	delete the object	remove the reference

### Exercise 3.27

What are the advantages or disadvantages of the methods of storing collections?

Point of discussion: Where are the libraries in my computer?

### Exercise 3.28

How does a C++ compiler find the libraries? Where are they stored in a computer?



End of Lecture 3