CS213/293 Data Structure and Algorithms 2024

Lecture 3: Stack and queue

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-08-03

Topic 3.1

Stack

Stack

Definition 3.1

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

- *Stack of papers in a copier*
- *Undo-redo features in editors*
- *Back button on Browser*

Interface of stack

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

Stack supports four interface methods

- `stack<T> s` : allocates 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

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

Exercise: action on the empty stack

Exercise 3.1

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.

Topic 3.2

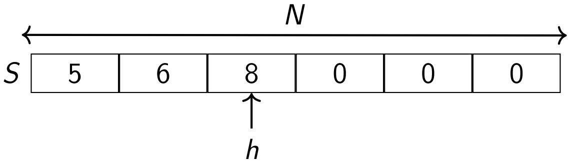
Implementing stack

Array-based stack

Let us look at a simplified array-based implementation of an array 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 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; //
  }
};
```

Efficiency

All operations are performed in $O(1)$ if there is no expansion to 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: `new_size = N + c` (for some fixed constant c)
- Exponential growth: `new_size = 2*N`

Which of the above two is better?

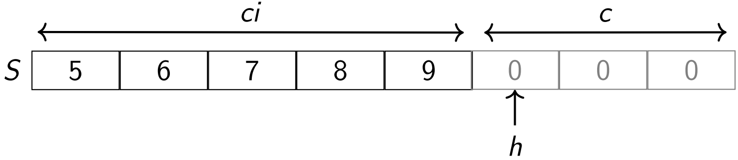
Analysis of constant growth

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

After every cth push, there will be an expansion operation.

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

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



Cost of ith 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:

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

Non-linear cost!

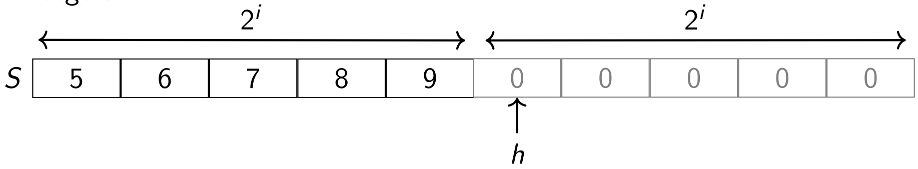
Analysis of exponential growth

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

The expansion operations will only occur at $2^i + 1$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}
- 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:

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

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

Exercise 3.2

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

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

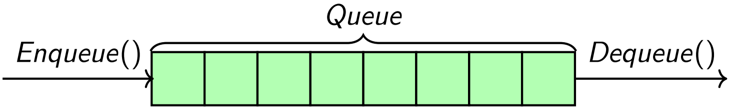
Queue

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 lift in a building (priority queue)*

Interface of queue

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

Queue supports four main interface methods

- `queue<T> q` : allocates 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 C++ library uses push for enqueue and pop uses for dequeue. Other languages such as Java uses the terms enqueue and dequeue.

Axioms of queue

1. `queue<T> q; q.enqueue(e); Assert(q.front() == e);`

2. `queue<T> q,q1; q.enqueue(e); q.dequeue(); Assert(q1 == q);`

3. ```
   q.enqueue(e1); Assume(q1 == q);
   q.enqueue(e2);
   Assert(q.front() == q1.front());
   ```

4. ```
   q.enqueue(e1); Assume(q1 == q);
   q.enqueue(e2);q.dequeue(); q1.dequeue();q1.enqueue(e2);
   Assert(q == q1);
   ```

Exercise 3.3

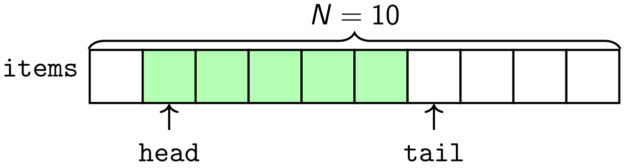
Why do the above four axioms define queue?

Topic 3.6

Array implementation of queue

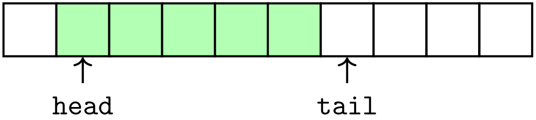
Array-based implementation

- 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

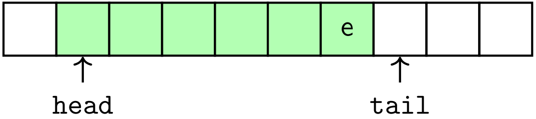


Enqueue operation on array

Consider the state of the queue

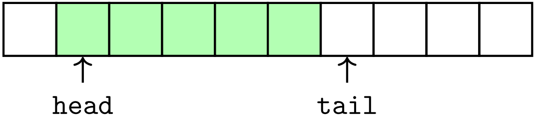


After the enqueue(e) operation:

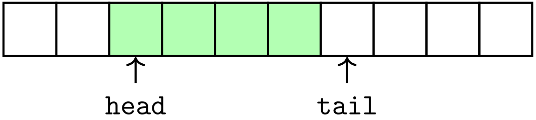


Dequeue operation on array

Consider the state of the queue



After dequeue() operation:

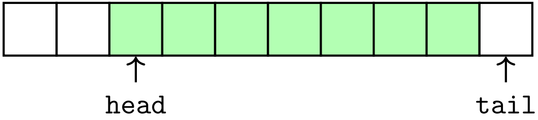


Exercise 3.4

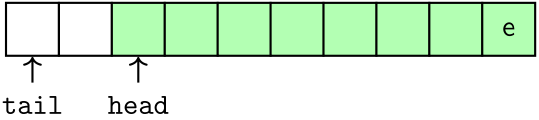
1. *Where will front() read from?*
2. *What is the size of the queue?*

Wrap around to utilize most of the array

Consider the state of the queue

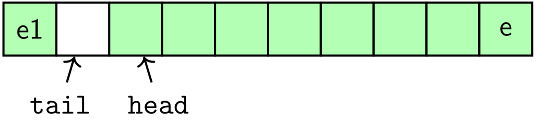


After enqueue(e) operation, we move the tail to 0.



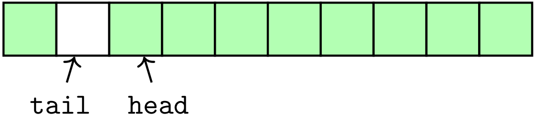
Wrap-around allows us to use the array repeatedly.

After another `enqueue(e1)` operation:

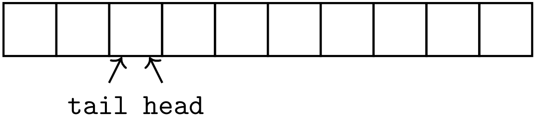


Full and empty queue

Full queue:



Empty queue:



Exercise 3.5

Can we use all N cells for storing elements?

Array implementation

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

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

Object items[N];                        //Some initial size

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

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

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

Array implementation

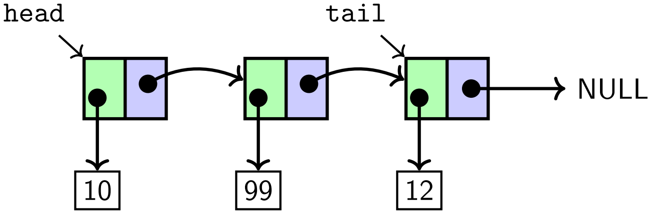
```
void dequeue() {
  if( empty() ) throw Empty;             // Queue is empty
  free(items[head]);items[head] = NULL; // Clear memory
  head = (head+1)%N;      //Remove an element
}


void enqueue( Object x ) {
  if ( size() == N-1 ) expand(); // Queue is full;expand
  items[tail] = x;
  tail = (tail+1)%N; // insert element
}
```

Topic 3.7

Queue via linked list

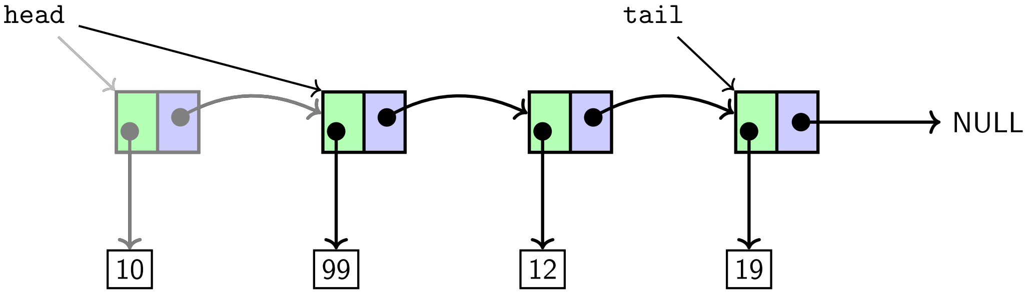
Linked lists

Definition 3.3
A linked list consists of nodes 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 3.6
If we use a linked list for implementing a queue, which side should be the front of the queue?

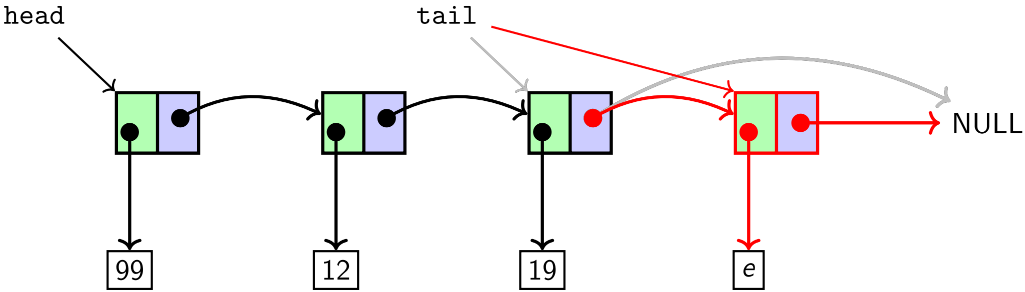
Dequeue in linked lists



Exercise 3.7

What happens to the object containing 10?

Enqueue(e) in linked lists



Exercise 3.8

a. Which one is better: array or linked list?

b. Do we need the tail pointer?

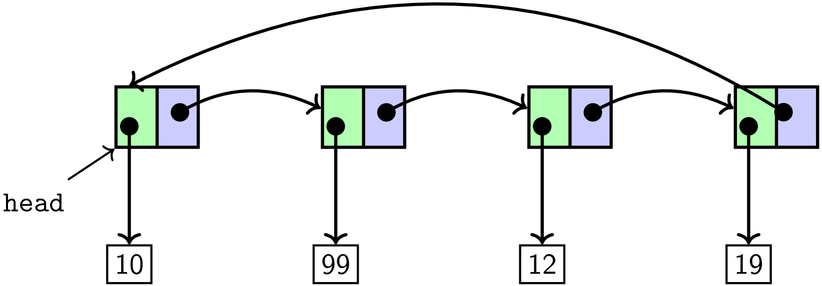
Topic 3.8

Circular linked list

Circular linked lists

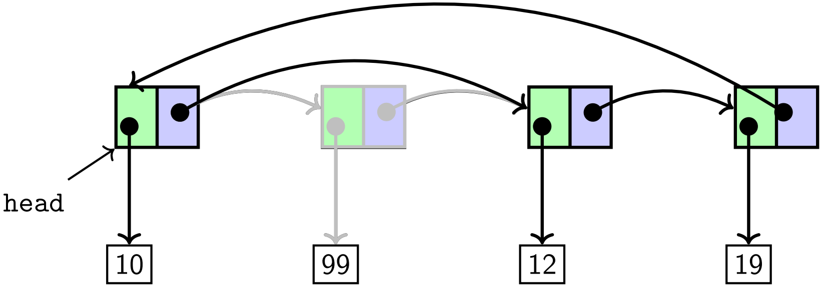
Definition 3.4

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

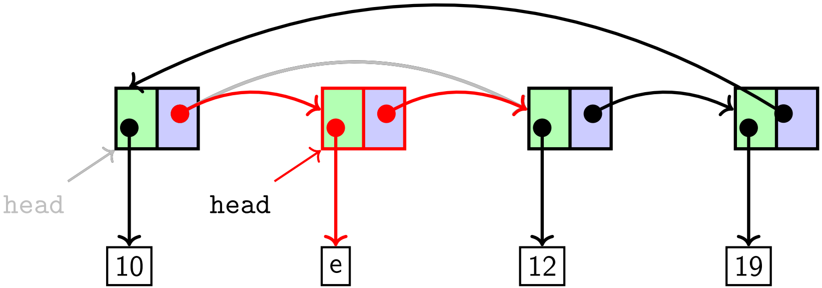


A head pointer points at some node of the circular list. A single pointer can do the job of the head and tail.

Dequeue in circular linked lists



enqueue(e) in circular linked lists



Exercise 3.9

a. Which element should be returned by front()?

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

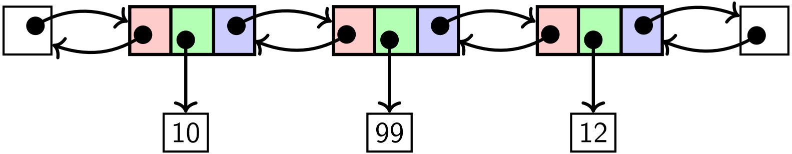
Topic 3.9

Dqueue via a doubly linked list

Doubly linked lists

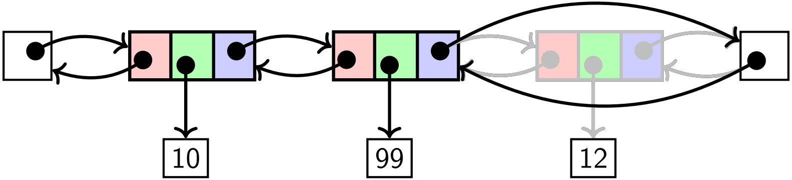
Definition 3.5

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 pointer. 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 used to store the start and end points of the list.

Deleting a node in a doubly linked list



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`

Queue supports four main interface methods

- `deque<T> q` : allocates new queue q
- `q.push_back(e)` : Adds the given element e to the back.
- `q.push_front(e)` : Adds the given element e to the front.
- `q.pop_front()` : Removes the first element from the queue.
- `q.pop_back()` : Removes the last element from the queue.
- `q.front()` : access the first element .
- `q.back()` : access the first element .

Some support functions

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

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

Stack and queue via Deque

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

Exercise 3.10

- *Which functions of deque implement stack?*
- *Which functions of deque implement queue?*

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

Exercise 3.11

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

Topic 3.10

Tutorial problems

Use of stack

Exercise 3.12

The span of a stock's price on ith day is the maximum number of consecutive days (up to ith 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 stack, 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.

Flipping Dosa

Exercise 3.13

There is a stack of dosas on a tava, of distinct radii. We want to serve the dosas of increasing radii. 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.

Exponential growth

Exercise 3.14

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 doubling policy?

b. Can we do a similar analysis for growth factor 1.5?

Problem: reversing a linked list

Exercise 3.15

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

Problem: middle element

Exercise 3.16

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

Stack and queue (Endsem 2023)

Exercise 3.17

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

Problem: messy queue

Exercise 3.18

The mess table queue problem: There is a common mess for k hostels. Each hostel has some $N_1,...,N_k$ 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 a normal queue? Would there be any difference in any statistic? If so, what?

Merge sorted queues (Quiz 2023)

Exercise 3.19

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

End of Lecture 3