CSC 212
Data Structures & Algorithms

Fall 2022 | Jonathan Schrader

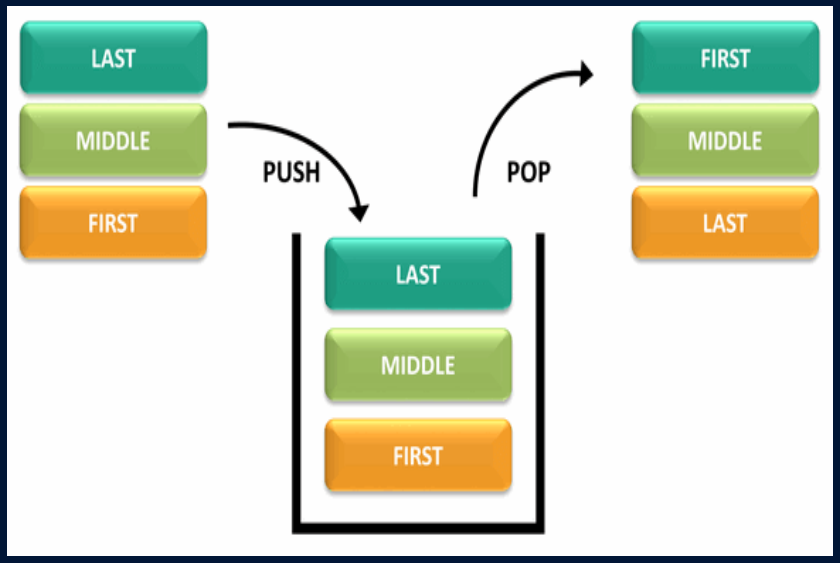
Stacks & Queues

Housekeeping

Assignment 2 Due

- Classes & Structs | CLASSES vs STRUCTS in C++ CLASSES in C++
- Pointers / References | POINTERS in C++ REFERENCES in C++
- Arrow operator & Dot Notation | The Arrow Operator in C++

Stacks



LIFO: Last In First Out

Documentation

std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The `std::stack` class is a container adaptor that gives the programmer the functionality of a stack - specifically, a LIFO (last-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

```cpp
#include <stack>
#include <iostream>

int main() {
  std::stack<int> s;
  s.push(1);
  s.push(2);
  s.push(3);
  std::cout << s.size() << " elements on stack\n";
  std::cout << "Top element: " << s.top() << "\n";
  std::cout << s.size() << " elements on stack\n";
  s.pop();
  std::cout << s.size() << " elements on stack\n";
  std::cout << "Top element: " << s.top() << "\n";
  return 0;
}
```

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

Member functions

(constructor)	constructs the `stack` (public member function)
(destructor)	destructs the `stack` (public member function)
`operator=`	assigns values to the container adaptor (public member function)

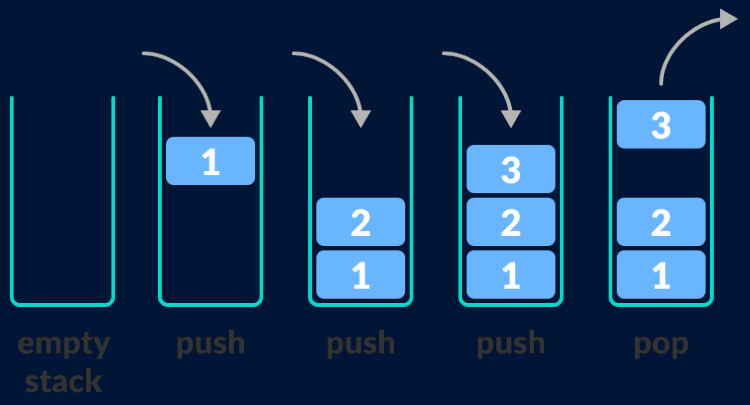
Element access

`top`	accesses the top element (public member function)

Capacity

`empty`	checks whether the underlying container is empty

Basic Operations



Push

- inserts one element onto the stack

Pop

- returns the element at the top of the stack (and removes it)

isEmpty

- not necessary, but sometimes useful

Implementation

Arrays

- *push* and *pop* at the end of the array (easier and efficient)
- can be *fixed-length*
- can also use a *dynamic array* (grows overtime)
  - additional cost for dynamic arrays

https://www.cs.usfca.edu/~galles/visualization/StackArray.html

Stack: Array Sample

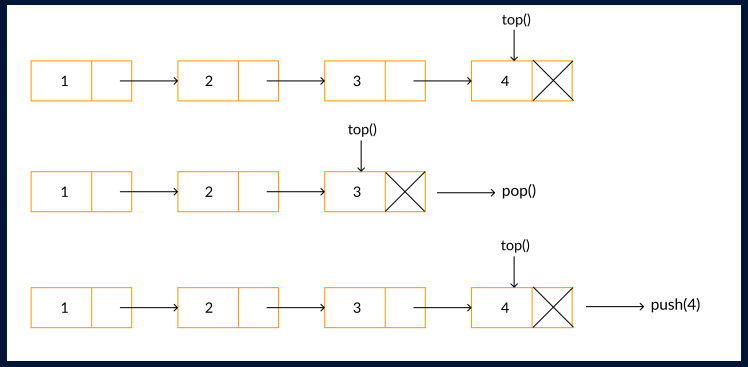
```cpp
// CPP program to illustrate
// Implementation of push() function

#include <iostream>
#include <stack>
using namespace std;

int main()
{
    // Empty stack
    stack<int> mystack;
    mystack.push(0);
    mystack.push(1);
    mystack.push(2);

    // Printing content of stack
    while (!mystack.empty()) {
        cout << ' ' << mystack.top();
        mystack.pop();
    }
}
```

https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/?ref=rp

Implementation

Linked Lists

- *push* and *pop* at front (could use the other end as well)



https://www.cs.usfca.edu/~galles/visualization/StackLL.html

Stack: LL Sample

```
void pop()
{
  Node* temp;
  if (top == NULL) {
    cout << "\nStack Underflow" << endl;
    exit(1);
  }
  else {
    temp = top;
    top = top->link;
    free(temp);
  }
}
```

```
void display()
{
  Node* temp;
  if (top == NULL) {
    cout << "\nStack Underflow";
    exit(1);
  }
  else {
    temp = top;
    while (temp != NULL) {
      cout << temp->data << "-> ";
      temp = temp->link;
    }
  }
}
```

```
void push(int data)
{
  Node* temp = new Node();
  if (!temp) {
      cout << "\nStack Overflow";
      exit(1);
  }
  temp->data = data;
  temp->link = top;
  top = temp;
}
```

```
int isEmpty()
{
  return top == NULL;
}

int peek()
{
  if (!isEmpty())
    return top->data;
  else
    exit(1);
}
```

https://www.geeksforgeeks.org/implement-a-stack-using-singly-linked-list/

Considerations

Underflow

- error can be thrown when calling *pop* on an empty stack

Overflow

- error can be thrown when calling *push* on a full stack (especially in fixed-length implementations)

Applications

Undo in software applications…

Stack in compilers/programming languages…

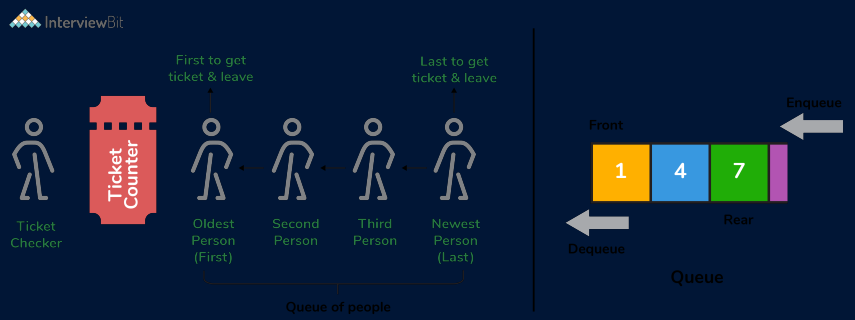
Parsing expressions…

etc…

Queues

FIFO: First In First Out

Basic Operations



Enqueue

- inserts one element onto the queue

Dequeue

- returns the element at the top of the queue (and removes it)

isEmpty

- not necessary, but sometimes useful

Documentation

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

The `std::queue` class is a container adaptor that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

```cpp
#include <stack>
#include <iostream>

int main() {
  std::queue<int> q1;
  q1.push(5);
  std::cout << q1.size() << "\n";

  std::queue<int> q2(q1);
  std::cout << s.size() << "\n";

  std::deque<int> deq { 3, 1, 4, 1, 5 };
  std::queue<int> q3(deq);
  std::cout << q3.size() << "\n";
}
```

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

Member functions

(constructor)	constructs the queue (public member function)
(destructor)	destructs the queue (public member function)
`operator=`	assigns values to the container adaptor (public member function)
Element access	
`front`	access the first element (public member function)
`back`	access the last element (public member function)
Capacity	
`empty`	checks whether the underlying container is empty

Implementation

Arrays

- *enqueue* and *dequeue* at different ends of the array (easier and efficient)
- can be *fixed-length*
- can also use a *dynamic array* (grows over time)
  - additional cost for dynamic arrays

https://www.cs.usfca.edu/~galles/visualization/QueueArray.html

Queue: Array Sample

```
void enqueue(Queue* queue, int item)
{
  if (isFull(queue))
    return;
  queue->rear = (queue->rear + 1)
                % queue->capacity;
  queue->array[queue->rear] = item;
  queue->size = queue->size + 1;
  cout << item << " enqueued to queue\n";
}
```

```
int dequeue(Queue* queue)
{
  if (isEmpty(queue))
    return INT_MIN;
  int item = queue->array[queue->front];
  queue->front = (queue->front + 1)
                 % queue->capacity;
  queue->size = queue->size - 1;
  return item;
}
```

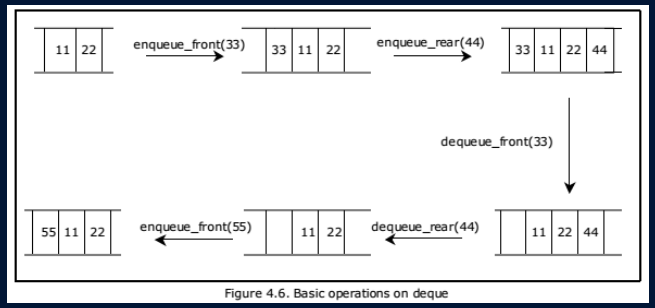


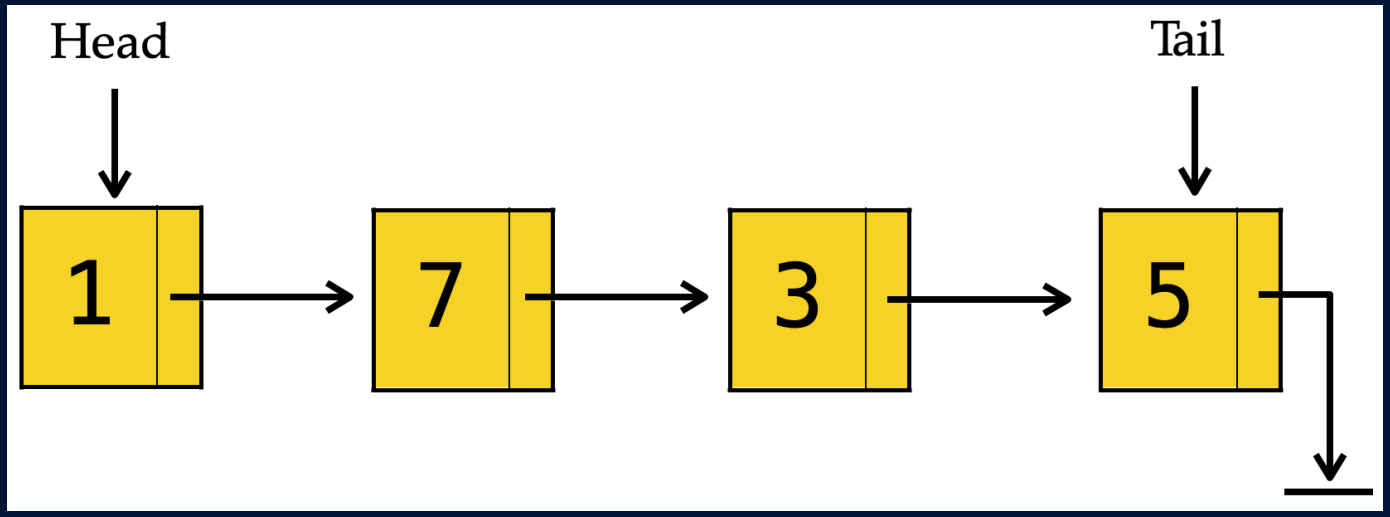
Figure 4.6. Basic operations on deque

https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/

Implementation

Linked Lists

- *enqueue* and *dequeue* at different ends



https://www.cs.usfca.edu/~galles/visualization/QueueArray.html

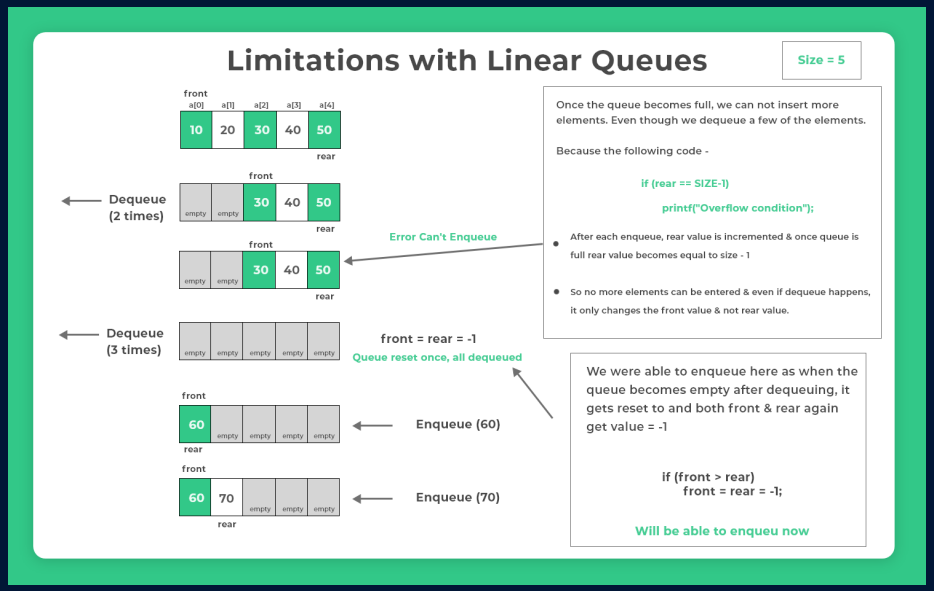
Queue: LL Sample

```
void enQueue(int x)
{
  QNode* temp = new QNode(x);

  if (rear == NULL) {
    front = rear = temp;
    return;
  }

  rear->next = temp;
  rear = temp;
}
```

```
void deQueue()
{
  if (front == NULL)
    return;

  QNode* temp = front;
  front = front->next;

  if (front == NULL)
    rear = NULL;

  delete (temp);
}
```

https://www.geeksforgeeks.org/queue-linked-list-implementation/

Considerations



Underflow

- error can be thrown when calling *dequeue* on an empty queue

Overflow

- error can be thrown when calling *enqueue* on a full queue (especially in fixed-length implementations)

Applications

Media Playlists (Youtube, Spotify, Music,etc.)

Process management in Operating Systems

Simulations

Used in other algorithms

etc…