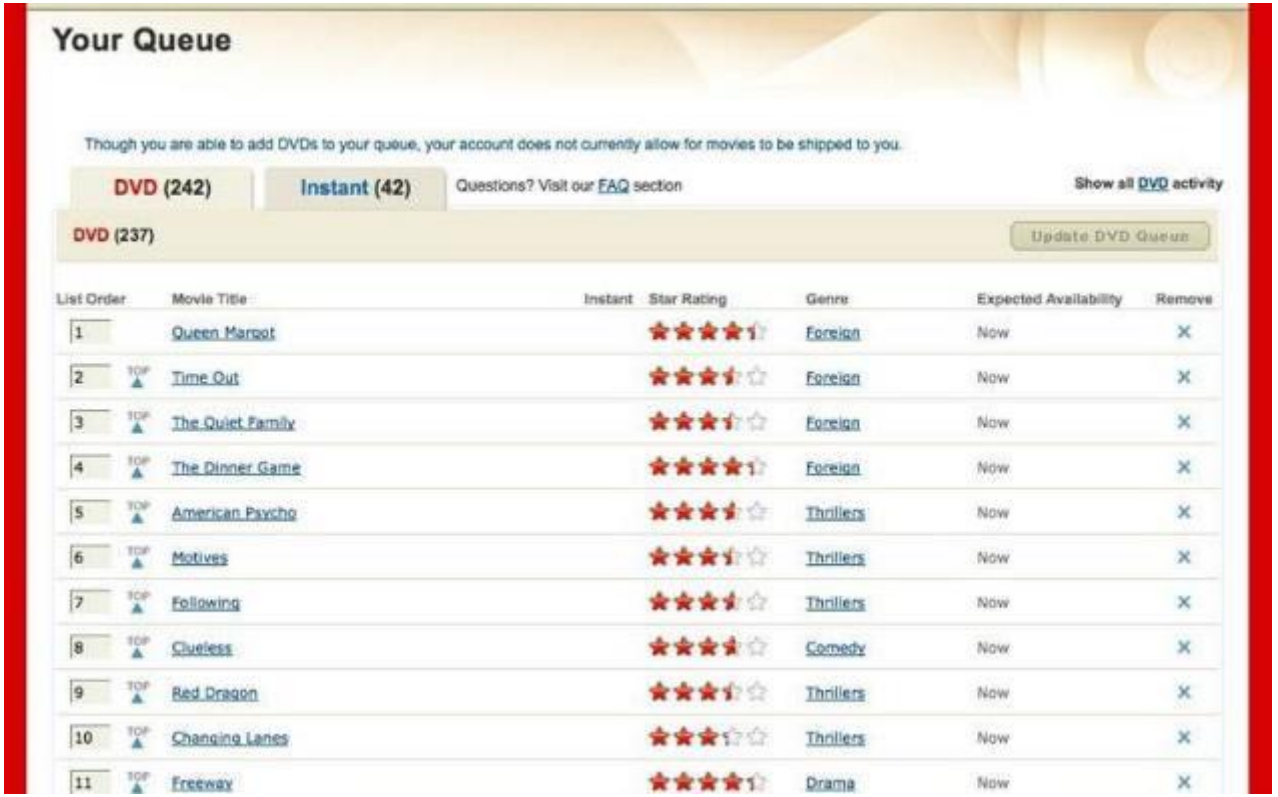
Lecture 2
Stacks and Queues



EECS 281: Data Structures & Algorithms

Data Structures and Abstract Data Types

Data Structures & Algorithms

Data Structures and ADTs

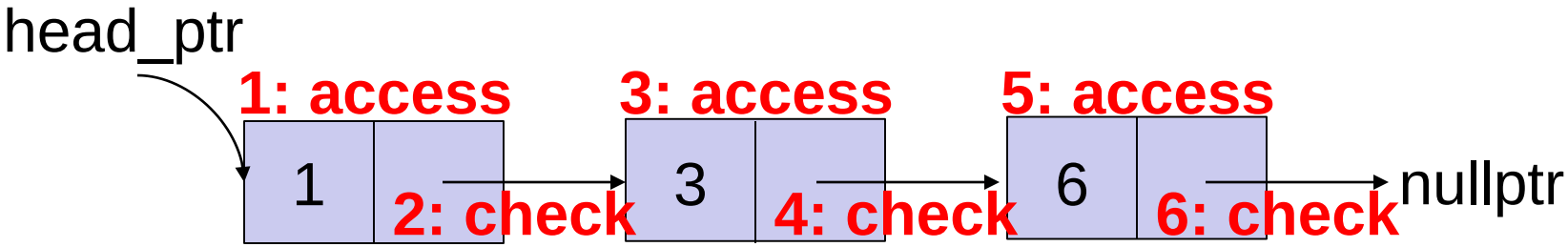
- Need a way to store and organize data in order to facilitate access and modifications
- An **abstract data type (ADT)** combines data with valid operations and their behaviors on stored data
  - e.g., insert, delete, access
  - ADTs define an interface
- A **data structure** provides a concrete implementation of an ADT

Measuring Performance

- Several design choices for implementing ADTs
  - Contiguous data (arrays or vectors)
  - Connected data (pointers or linked lists/trees)
- Runtime speed and size of data structure
  - How much time is needed to perform an operation? (count number of steps)
  - How much space is needed to perform an operation? (count size of data and pointers/metadata)
  - How does size/number of inputs affect these results? (constant, linear, exponential, etc.)
- We formalize performance measurements with **complexity analysis**

Analysis Example

- How many operations are needed to insert a value at the end of this singly-linked list?



- Can you generalize this for a list with ***n*** elements?

7: Create Node
8: Insert Value
9: Update Pointer

2n + 3

Linear function is important—coefficients and constants don't matter much

Choosing a Data Structure for a Given Application

- What to look for
  - The right operations (e.g., add_elt, remove_elt)
  - The right behavior (e.g., push_back, pop_back)
  - The right trade-offs for runtime complexities
  - Memory overhead

- Potential concern
  - Limiting interface to avoid problems (e.g., no insert_mid)
- Examples
  - **Order tracking at a fast-food drive-through** (pipeline)
  - **Interrupted phone calls to a receptionist**
  - **Your TODO list**

Data Structures and Abstract Data Types

Data Structures & Algorithms

Basic Containers: Stack

Data Structures & Algorithms

Stack ADT: Interface

- Supports insertion/removal in LIFO order
  - Last In, First Out

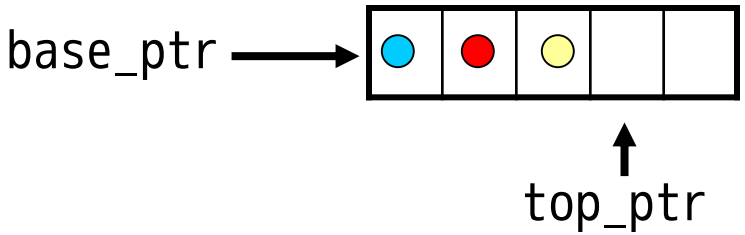
Method	Description
`push(object)`	Add object to top of the stack
`pop()`	Remove top element
`object &top()`	Return a reference to top element
`size()`	Number of elements in stack
`empty()`	Checks if stack has no elements

Examples

- Web browser's "back" feature
- Text editor's "Undo" feature
- Function calls in C++

Stack: Implementation – Array/Vector

Keep a pointer (`top_ptr`) just past the last element

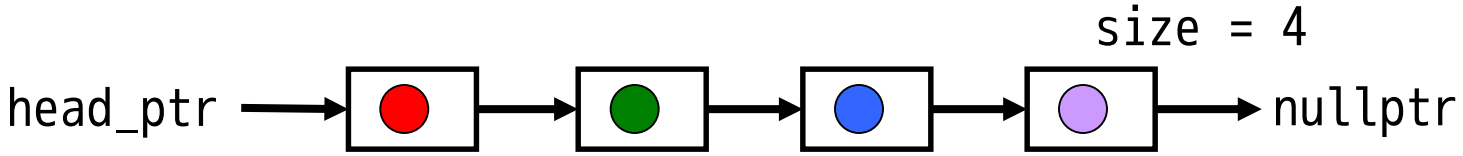


Method	Implementation
`push(object)`	1. If needed, allocate a bigger array and copy data 2. Add new element at top_ptr, increment top_ptr
`pop()`	Decrement `top_ptr`
`object &top()`	Dereference `top_ptr - 1`
`size()`	Subtract `base_ptr` from `top_ptr` pointer
`empty()`	Check if `base_ptr == top_ptr`

How many steps/operations for each method?

Stack: Implementation – Linked List

Singly-linked is sufficient



Method	Implementation
`push(object)`	Insert new node at `head_ptr`, increment size
`pop()`	Delete node at `head_ptr`, decrement size
`object &top()`	Dereference `head_ptr`
`size()`	Return `size`
`empty()`	Check if `size == 0` or `head_ptr == nullptr`

*Alternative approach: eliminate `size`, count nodes each time

How many steps/operations for each method?

Is an array or linked list more efficient for stacks?

Stack: Which Implementation?

Method	Array/Vector	Linked List
`push(object)`	Constant (linear when resizing vector)*	Constant
`pop()`	Constant	Constant
`object &top()`	Constant	Constant
`size()`	Constant	Constant (with tracked size)
`empty()`	Constant	Constant

*Averages out to constant with many pushes (amortized constant)

- The asymptotic complexities of each are similar
- The constant factor attached to the complexity is lower for vector
  - Constant number of operations, but there is “less” to do
  - The linked list must allocate memory for each node individually!
- The linked list also has higher memory overhead
  - i.e. Pointers between nodes as well as the actual data payload

STL Stacks: std::stack<>

- Code: `#include <stack>`
- You can choose the underlying container
- All operations are implemented generically on top of the given container
  - No specialized code based on given container

	Stack
Default Underlying Container	`std::deque<>`
Optional Underlying Container	`std::list<>`† `std::vector<>`

†`std::list<>` is a doubly-linked list

Basic Containers: Stack

Data Structures & Algorithms

Basic Containers: Queue

Data Structures & Algorithms

Queue ADT: Interface

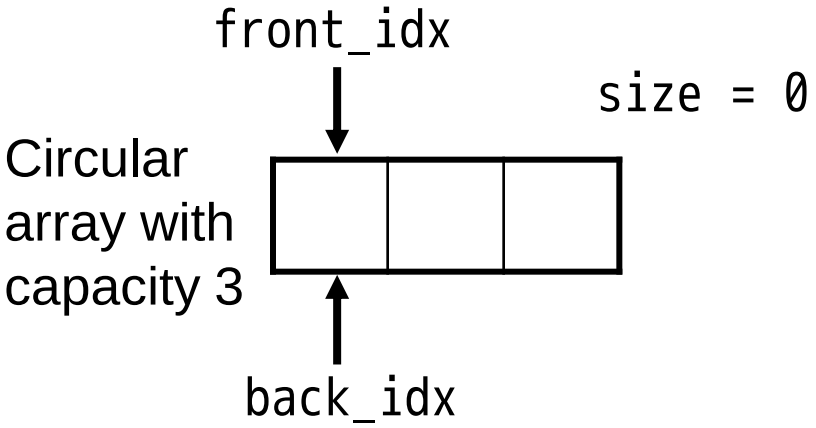
- Supports insertion/removal in FIFO order
  - First In, First Out

Method	Description
`push(object)`	Add element to back of queue
`pop()`	Remove element at front of queue
`object &front()`	Return reference to element at front of queue
`size()`	Number of elements in queue
`empty()`	Checks if queue has no elements

Examples

- Waiting in line for lunch
- Adding songs to the end of a playlist

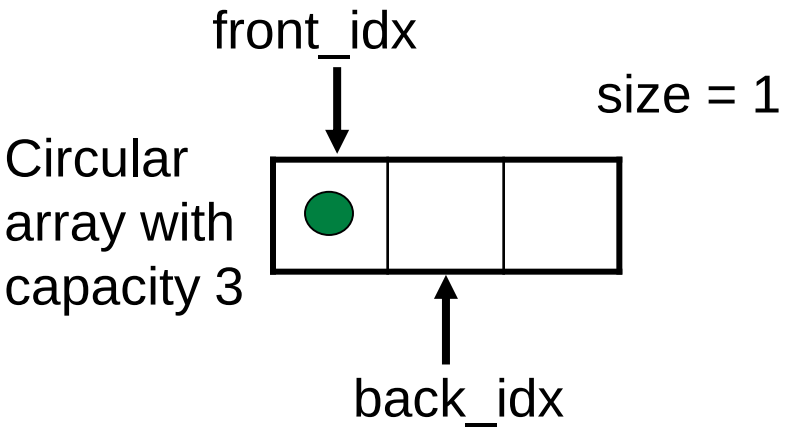
Queue: Implementation – Circular Buffer



Event Sequence

1. `back_idx == front_idx`
   since array is empty

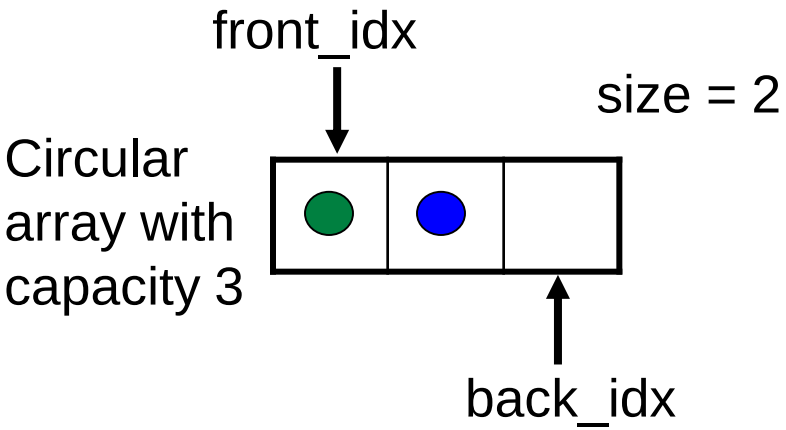
Queue: Implementation – Circular Buffer



Event Sequence

1. back_idx == front_idx since array is empty
2. push element

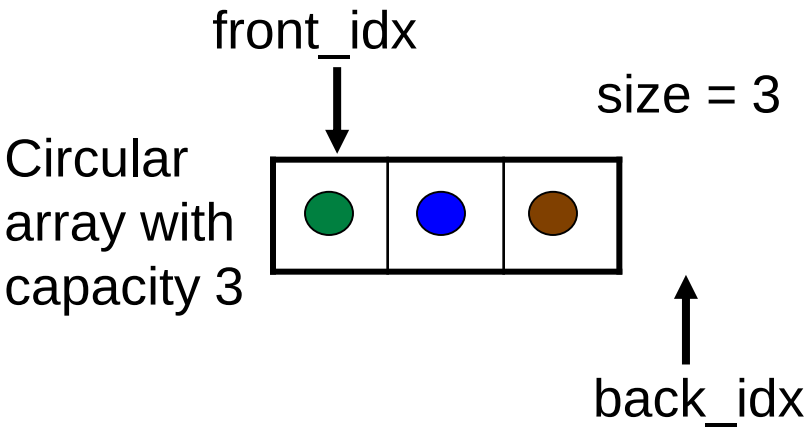
Queue: Implementation – Circular Buffer



Event Sequence

1. back_idx == front_idx since array is empty
2. push element
3. push element

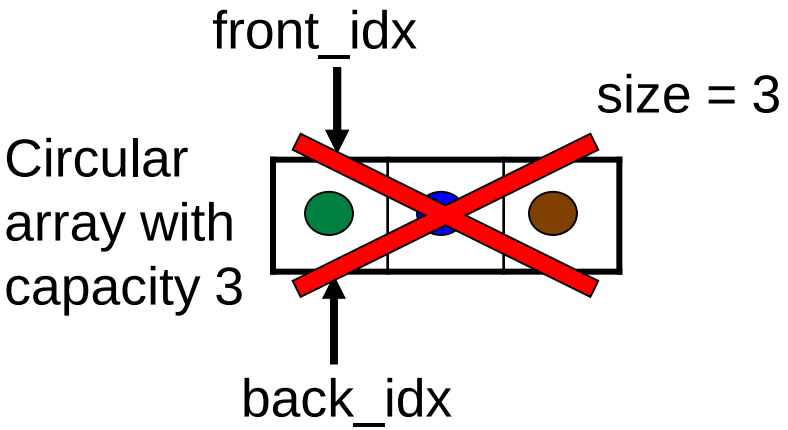
Queue: Implementation – Circular Buffer



Event Sequence

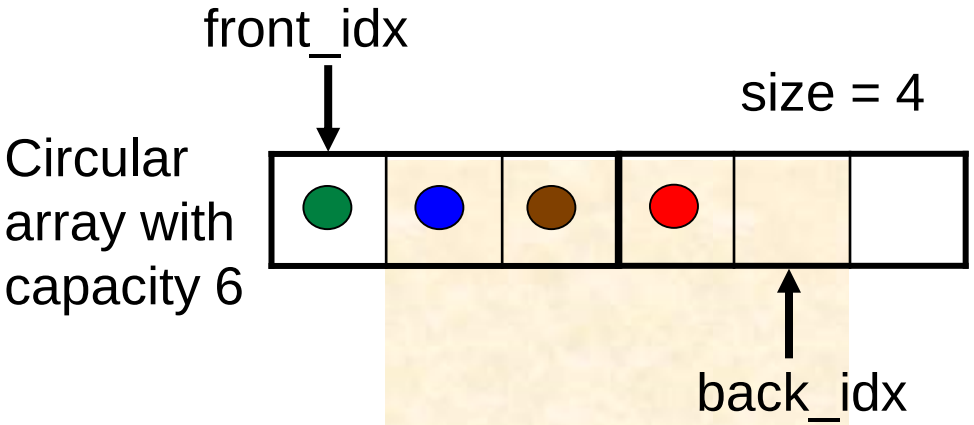
1. back_idx == front_idx since array is empty
2. push element
3. push element
4. push element

Queue: Implementation – Circular Buffer



Event Sequence

1. back_idx == front_idx since array is empty
2. push element
3. push element
4. push element
5. push element (need to allocate more memory)*

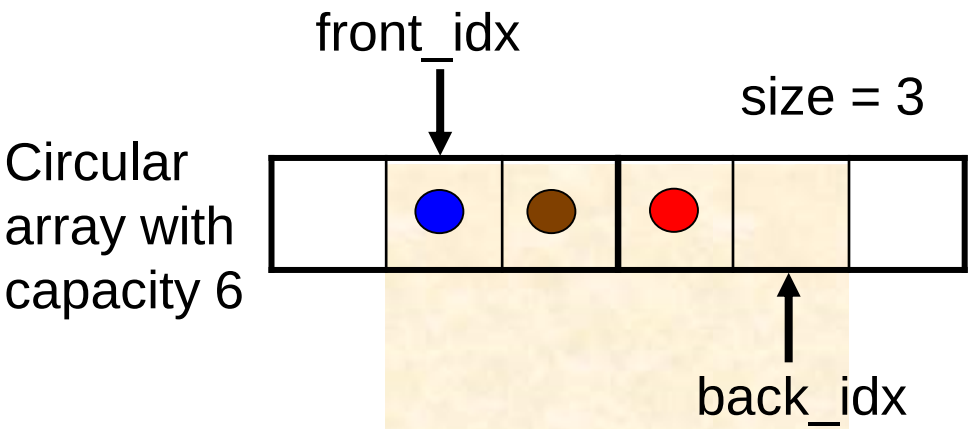


* When allocating more memory, it is common to double memory

Queue: Implementation – Circular Buffer

Event Sequence

1. back_idx == front_idx since array is empty
2. push element
3. push element
4. push element
5. push element (need to allocate more memory)*
6. pop element

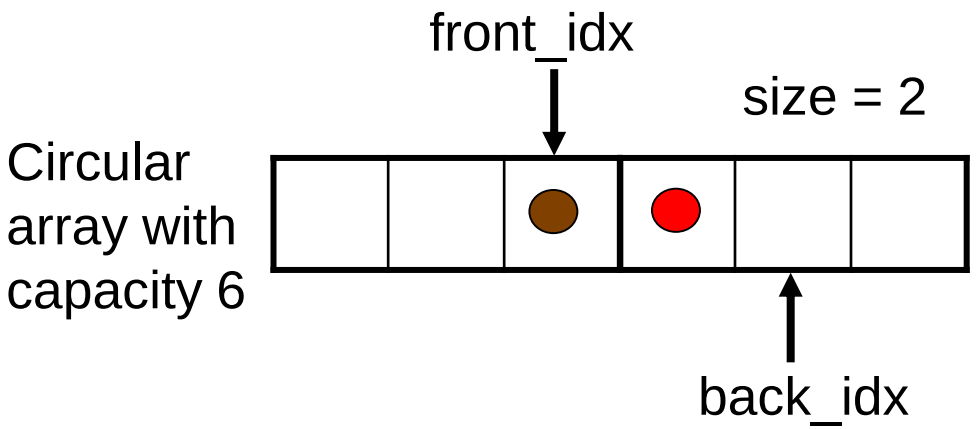


* When allocating more memory, it is common to double memory

Queue: Implementation – Circular Buffer

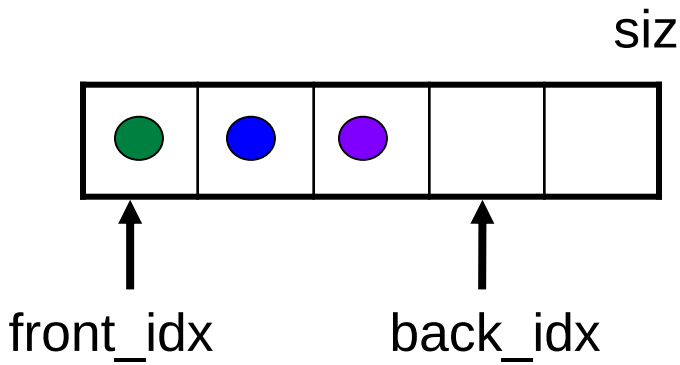
Event Sequence

1. back_idx == front_idx since array is empty
2. push element
3. push element
4. push element
5. push element (need to allocate more memory)*
6. pop element
7. pop element



Queue: Implementation – Circular Buffer

Use a circular array

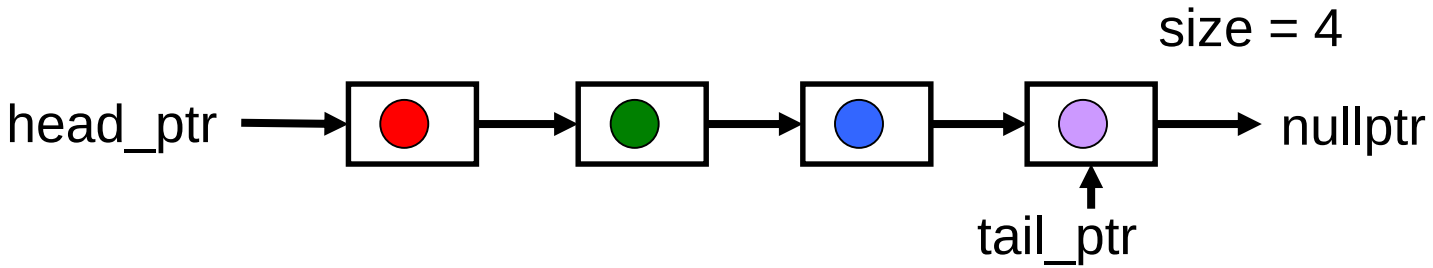


Method	Implementation
push(object)	1. If size == capacity, reallocate larger array and copy over elements, "unrolling" as you go unroll: start front_idx at 0, insert all elements 2. Insert value at back_idx, increment size and back_idx, wrapping around either as needed
pop()	Increment front_idx, decrement size
object &front()	Return reference to element at front_idx
size()	Return size
empty()	Check if size == 0

How many steps/operations for each method?

Queue: Implementation – Linked List

Singly-linked is sufficient



Method	Implementation
push(object)	Append node after tail_ptr, increment size
pop()	Delete node at head_ptr, decrement size
object &front()	Deference head_ptr
size()	Return size
empty()	Return head_ptr == nullptr

*Alternative approach: count nodes when needed

How many steps/operations for each method?

Queue: Which Implementation?

Method	Array/Vector	Linked List
push(object)	Constant (linear when resizing vector)*	Constant
pop()	Constant	Constant
object &front()	Constant	Constant
size()	Constant	Constant (with tracked size)
empty()	Constant	Constant

*Averages out to constant with many pushes (amortized constant)

- The asymptotic complexities of each are similar
- The constant factor attached to the complexity is lower for vector
  - Constant number of operations, but there is “less” to do
  - The linked list must allocate memory for each node individually!
- The linked list also has higher memory overhead
  - i.e. Pointers between nodes as well as the actual data payload

STL Queues: std::queue<>

- Code: #include <queue>
- You can choose the underlying container
- All operations are implemented generically on top of the given container
  - No specialized code based on given container

	Queue
Default Underlying Container	std::deque<>
Optional Underlying Container	std::list<>

Basic Containers: Queue

Data Structures & Algorithms

Basic Containers: Deque

Data Structures & Algorithms

Deque Terminology Clarification

- "Deque" is an abbreviation of Double-Ended Queue.
  Pronounced "deck"
- "Dequeue" is another name for removing something from a queue.
  Pronounced "dee-queue"
- The STL includes std::deque<>, which is an implementation of a Deque, and is usually based on a growable collection of fixed-sized arrays.

Deque ADT: a queue and stack in one (Double-ended Queue)

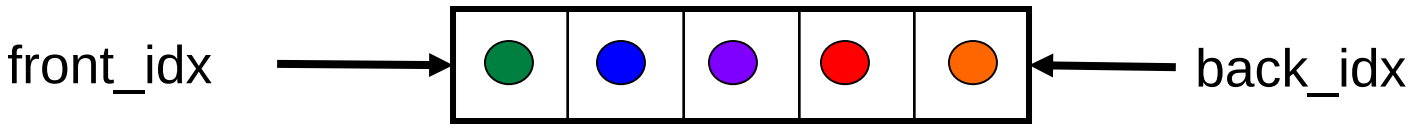
- ADT that allows efficient insertion and removal from the front and the back
- 6 major methods
  - push_front(), pop_front(), front()
  - push_back(), pop_back(), back()
- Minor methods
  - size(), empty()
- Can traverse using iterator

- STL incudes constant time operator[]()

Simple Deque Implementation

Circular Buffer

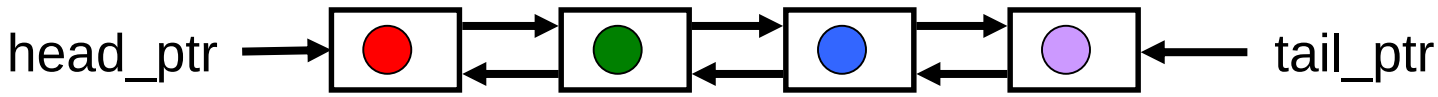
– front_idx and back_idx both get incremented/decremented



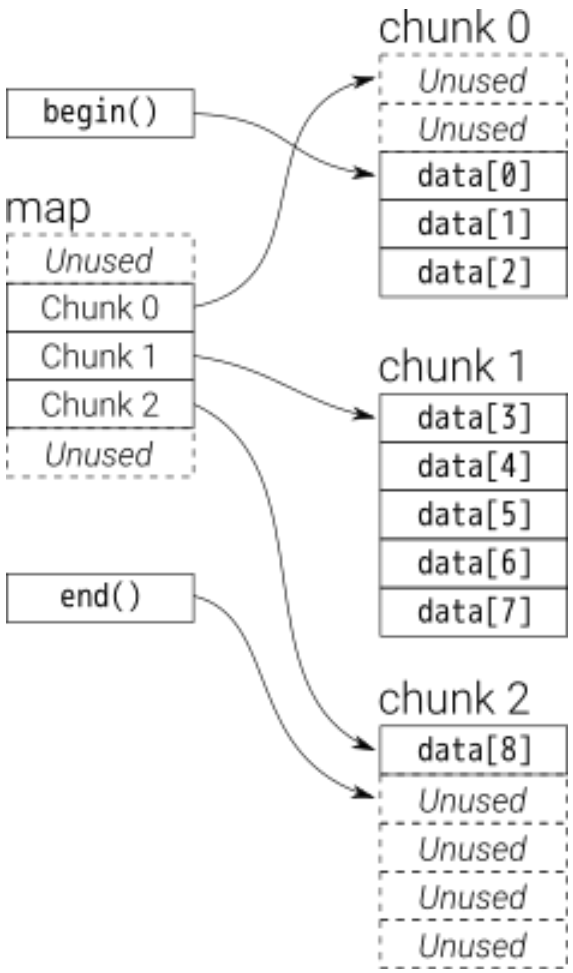
Doubly-linked list

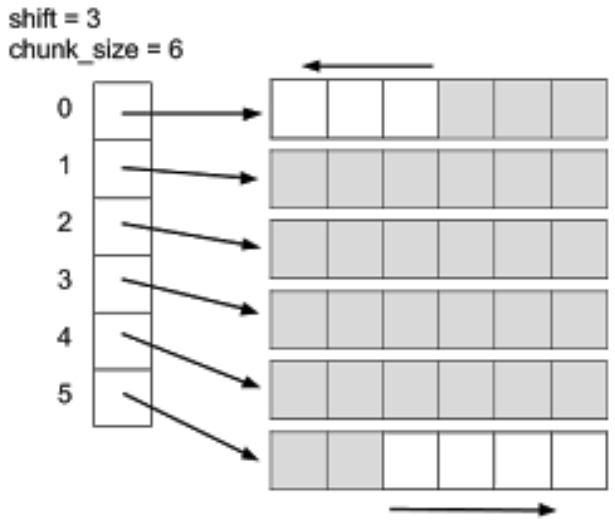
– Singly-linked doesn't support efficient removal

– Other operations map directly to doubly-linked list operations



STL Deque, Two Internal Views





Taken from: https://stackoverflow.com/questions/6292332/what-really-is-a-deque-in-stl and http://cpp-tip-of-the-day.blogspot.com/2013/11/how-is-stddeque-implemented.html

STL Deques: std::deque<>

- Code: #include <deque>
- Stack/Queue-like behavior at both ends
- Random access with [] or .at()

Basic Containers: Deque

Data Structures & Algorithms

Customizable Containers: Priority Queue

Data Structures & Algorithms

What is a Priority Queue?

- Each datum paired with a priority value
  - Priority values are usually numbers
  - Should be able to compare priority values (<)
- Supports insertion of data and inspection
- Supports removal of datum with highest priority
  - ”Most important” determined by given ordering

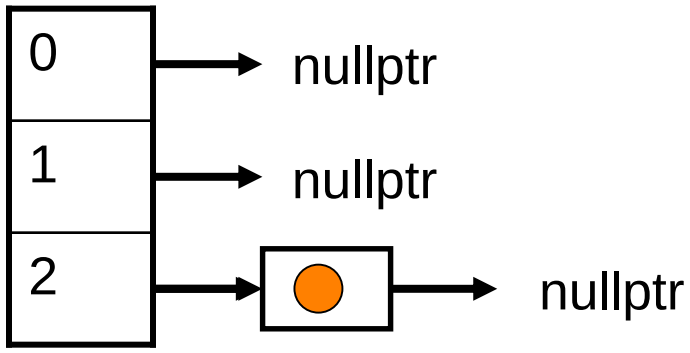
Like a group of bikers where the fastest ones exit the race first

Priority Queue Example: Emergency Call Center

- Operators receive calls and assign levels of urgency
- Lower numbers indicate more urgent calls
- Calls are dispatched (or not dispatched) by computer to police squads based on urgency

---

1. Level 2 call comes in

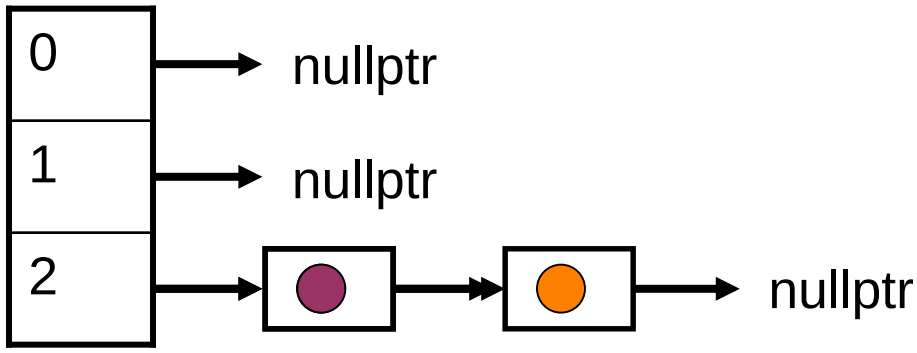


Priority Queue Example: Emergency Call Center

- Operators receive calls and assign levels of urgency
- Lower numbers indicate more urgent calls
- Calls are dispatched (or not dispatched) by computer to police squads based on urgency

---

1. Level 2 call comes in
2. Level 2 call comes in

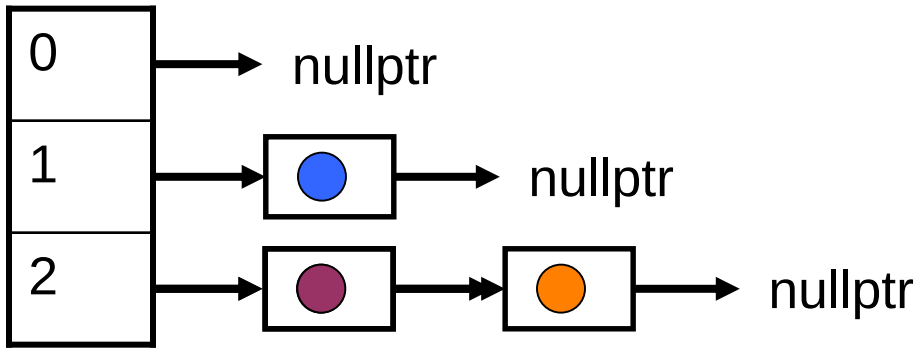


Priority Queue Example: Emergency Call Center

- Operators receive calls and assign levels of urgency
- Lower numbers indicate more urgent calls
- Calls are dispatched (or not dispatched) by computer to police squads based on urgency

---

1. Level 2 call comes in
2. Level 2 call comes in
3. Level 1 call comes in

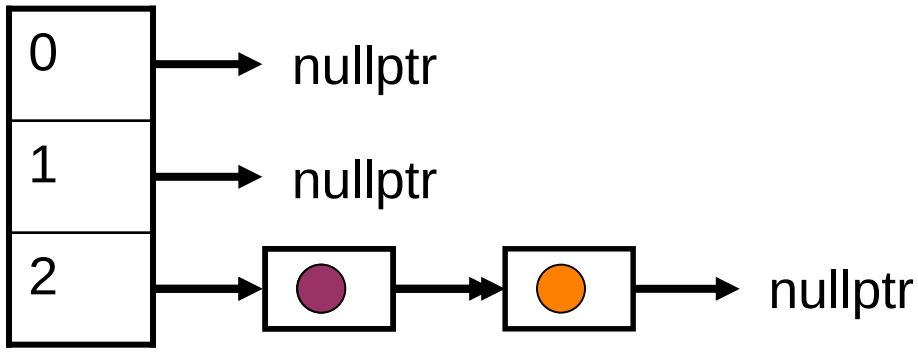


Priority Queue Example: Emergency Call Center

- Operators receive calls and assign levels of urgency
- Lower numbers indicate more urgent calls
- Calls are dispatched (or not dispatched) by computer to police squads based on urgency

---

1. Level 2 call comes in
2. Level 2 call comes in
3. Level 1 call comes in
4. A call is dispatched

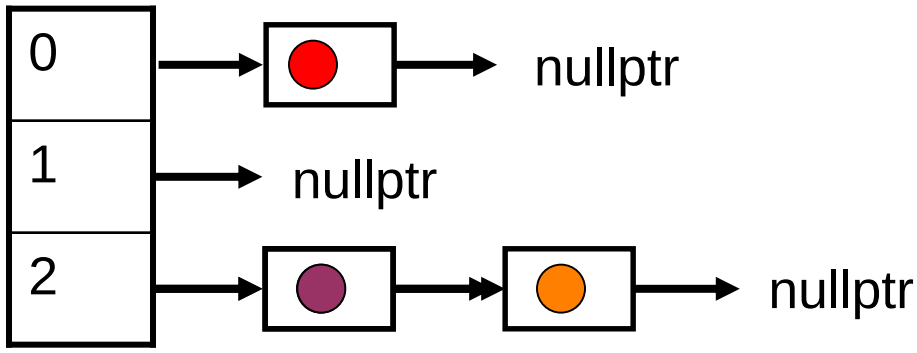


Priority Queue Example: Emergency Call Center

- Operators receive calls and assign levels of urgency
- Lower numbers indicate more urgent calls
- Calls are dispatched (or not dispatched) by computer to police squads based on urgency

---

1. Level 2 call comes in
2. Level 2 call comes in
3. Level 1 call comes in
4. A call is dispatched
5. Level 0 call comes in

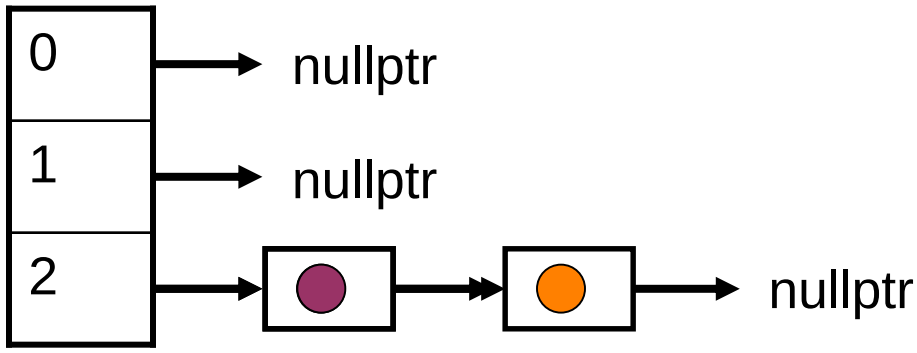


Priority Queue Example: Emergency Call Center

- Operators receive calls and assign levels of urgency
- Lower numbers indicate more urgent calls
- Calls are dispatched (or not dispatched) by computer to police squads based on urgency

---

1. Level 2 call comes in
2. Level 2 call comes in
3. Level 1 call comes in
4. A call is dispatched
5. Level 0 call comes in
6. A call is dispatched



Priority Queue ADT: Interface

- Supports insertion, with removal in descending priority order

Method	Description
push(object)	Add object to the priority queue
pop()	Remove highest priority element
const object &top()	Return a reference to highest priority element
size()	Number of elements in priority queue
empty()	Checks if priority queue has no elements

Examples

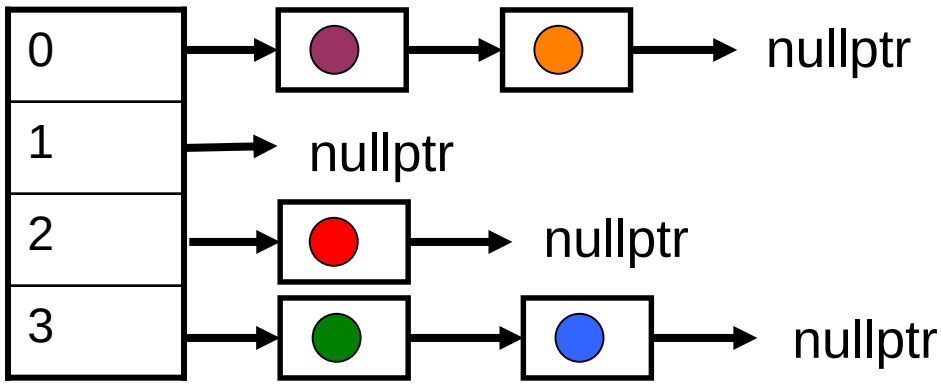
- Hospital queue for arriving patients
- Load balancing on servers

Priority Queue Implementations

	Insert	Remove
Unordered sequence container	Constant	Linear
Sorted sequence container	Linear	Constant
Heap	Logarithmic	Logarithmic
Array of linked lists (for priorities of small integers)	Constant	Constant

Array of Linked Lists

Priority value used as index value in array



A Customizable Container

- By default std::priority_queue<> uses std::less<> to determine relative priority of two elements
- A ”default PQ” is a “max-PQ”, where the largest element has highest priority
- If a “min-PQ” is desired, customize with std::greater<>, so the smallest element has highest priority
- If the PQ will hold elements that cannot be compared with std::less<> or std::greater<>, customize with custom comparator (function object)
- Custom comparators can work with objects, perform tie-breaks on multiple object members, and other functionality

std::priority_queue<>

- STL will maintain a Heap in any random access container
  - #include <queue>
- Common std::priority_queue<> declarations
  - “Max” PQ using std::less<>
    std::priority_queue<T> myPQ;
  - PQ using a custom comparator type, COMP
    std::priority_queue<T, vector<T>, COMP> myPQ;
- Manual priority queue implementation with standard library functions
  - #include <algorithm>
  - std::make_heap()
  - std::push_heap()
  - std::pop_heap()

Customizable Containers: Priority Queue

Data Structures & Algorithms

Generating Permutations

Data Structures & Algorithms

Algorithm Engineering: *Juggling with Stacks and Queues*

- Task: given *N* elements, generate all *N*-element permutations
- Ingredients of a solution
  - One recursive function
  - One stack
  - One queue
- Technique: moving elements between the two containers

Permutations Example

Input: {1, 2, 3}

Output: {

    {1, 2, 3},

    {1, 3, 2},

    {2, 3, 1},

    {2, 1, 3},

    {3, 1, 2},

    {3, 2, 1}

}

Implementation: Helper Function

```
1   template <typename T>
2   ostream &operator<<(ostream &out, const stack<T> &s) {
3     // display the contents of a stack on a single line
4     // e.g., cout << my_stack << endl;
5     stack<T> tmpStack = s;  // deep copy
6     while (!tmpStack.empty()) {
7       out << tmpStack.top() << ' ';
8       tmpStack.pop();
9     }  // while
10    return out;
11  }  // operator<<()
```

Better Helper Function

```
1  template <typename T>
2  ostream &operator<<(ostream &out, const vector<T> &v) {
3    // display the contents of a vector on a single line
4    // e.g., cout << my_vec << endl;
5    for (auto &el : v) {
6      out << el << ' ';
7    }  // for
8    return out;
9  }  // operator<<()
```

Implementation

```
1   template <typename T>
2   void genPerms(vector<T> &perm, deque<T> &unused) {
3     // perm is only a "prefix" until unused is empty
4     if (unused.empty()) {
5       cout << perm << '\n';
6       return;
7     }  // if
8     for (size_t k = 0; k != unused.size(); ++k) {
9       perm.push_back(unused.front());
10      unused.pop_front();
11      genPerms(perm, unused);
12      unused.push_back(perm.back());
13      perm.pop_back();
14    }  // for
15  }  // genPerms()
```

Implementation: Sample Driver

```
1  int main() {
2    size_t n;
3    cout << "Enter n: " << flush;
4    while (!(cin >> n)) {  // keep going while NO integer
5      cin.clear();
6      cin.ignore(numeric_limits<streamsize>::max(), '\n');
7      cout << "Enter n: " << flush;
8    }  // while
9
10   vector<size_t> perm;
11   deque<size_t> unused(n);
12   iota(unused.begin(), unused.end(), 1);
13   genPerms(perm, unused);
14   return 0;
15 }  // main()
```

Implement to Test

Q: how does the genPerms() compare to STL's function std::next_permutation() ?

http://en.cppreference.com/w/cpp/algorithm/next_permutation

A: each method has its advantages and can be more appropriate in some situations

Another Implementation

```
1   template <typename T>
2   void genPerms(vector<T> &path, size_t permLength) {
3     if (permLength == path.size()) {
4       // Do something with the path
5       return;
6     }  // if
7     if (!promising(path, permLength))
8       return;  // this partial permutation isn't better
9     for (size_t i = permLength; i < path.size(); ++i) {
10      swap(path[permLength], path[i]);
11      genPerms(path, permLength + 1);
12      swap(path[permLength], path[i]);
13    }  // for
14  }  // genPerms()
```

Generating Permutations

Data Structures & Algorithms