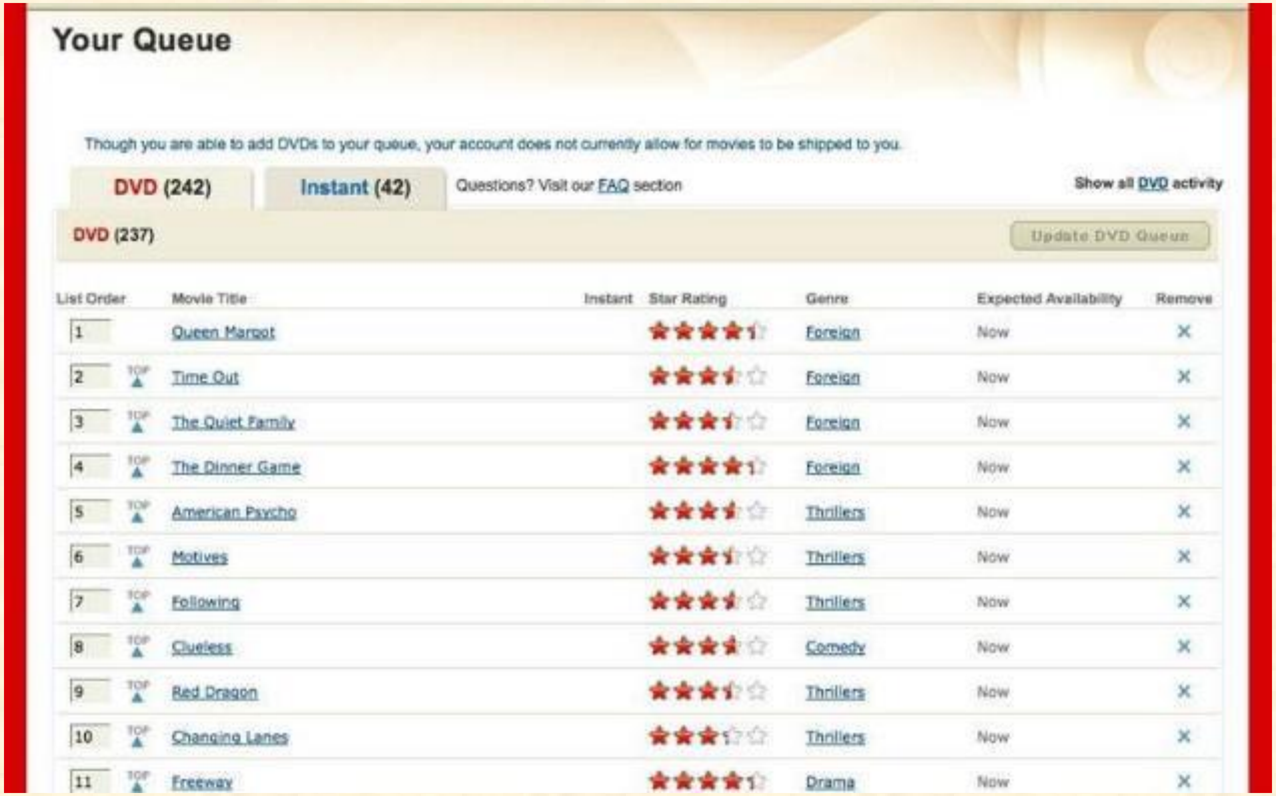
Lecture 7
Stacks and Queues



EECS 281: Data Structures & Algorithms

The Stack Container

- Supports insertion/removal in LIFO order

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 Example: Web Browsing

1. Open Browser to http://www.google.com
2. Search for “STL”
3. Go to SGI STL Guide
4. Click on the Table of Contents
5. Go to the stack page
6. Go back to the Table of Contents
7. Go to the basic_string page
8. Finished,close browser

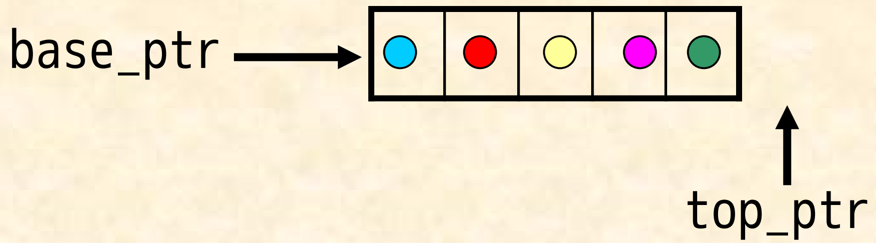
STL Stacks, Strings
Table of Contents
SGI STL Guide
Search Results: STL
Google Homepage

URL Stack

Should we use arrays or linked lists to implement stacks?

Stacks Using Arrays

Keep a pointer (`top_ptr`) to the last element of array

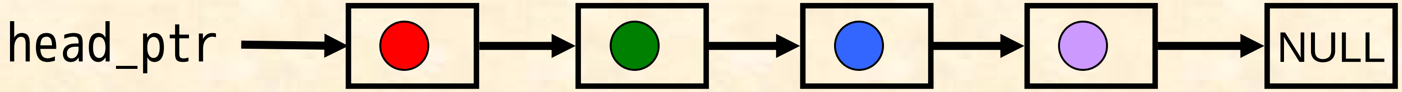


Method	Implementation
`push(object)`	Add new element at, and then increment `top_ptr` Allocate more space if necessary (requires copying)
`pop()`	Decrement `top_ptr`.
`object &top()`	Dereference `top_ptr - 1`.
`size()`	Subtract `base_ptr` from `top_ptr` pointer.
`empty()`	Are `base_ptr` and `top_ptr` equal?

What is the asymptotic runtime of each method?

Stacks Using Linked Lists

Singly-linked is sufficient



Method	Implementation
`push(object)`	Prepend node to list
`pop()`	Delete head node of list
`object &top()`	Return reference to data in head node
`size()`	Use existing `LinkedList::size()` method Be careful: `size()` in STL `<slist>` takes *O*(*n*) time (it computes size from scratch every time)
`empty()`	Use existing `LinkedList::empty()` method

What is the asymptotic runtime of each method?

Is an array or linked list more efficient for stacks?

The Queue Container

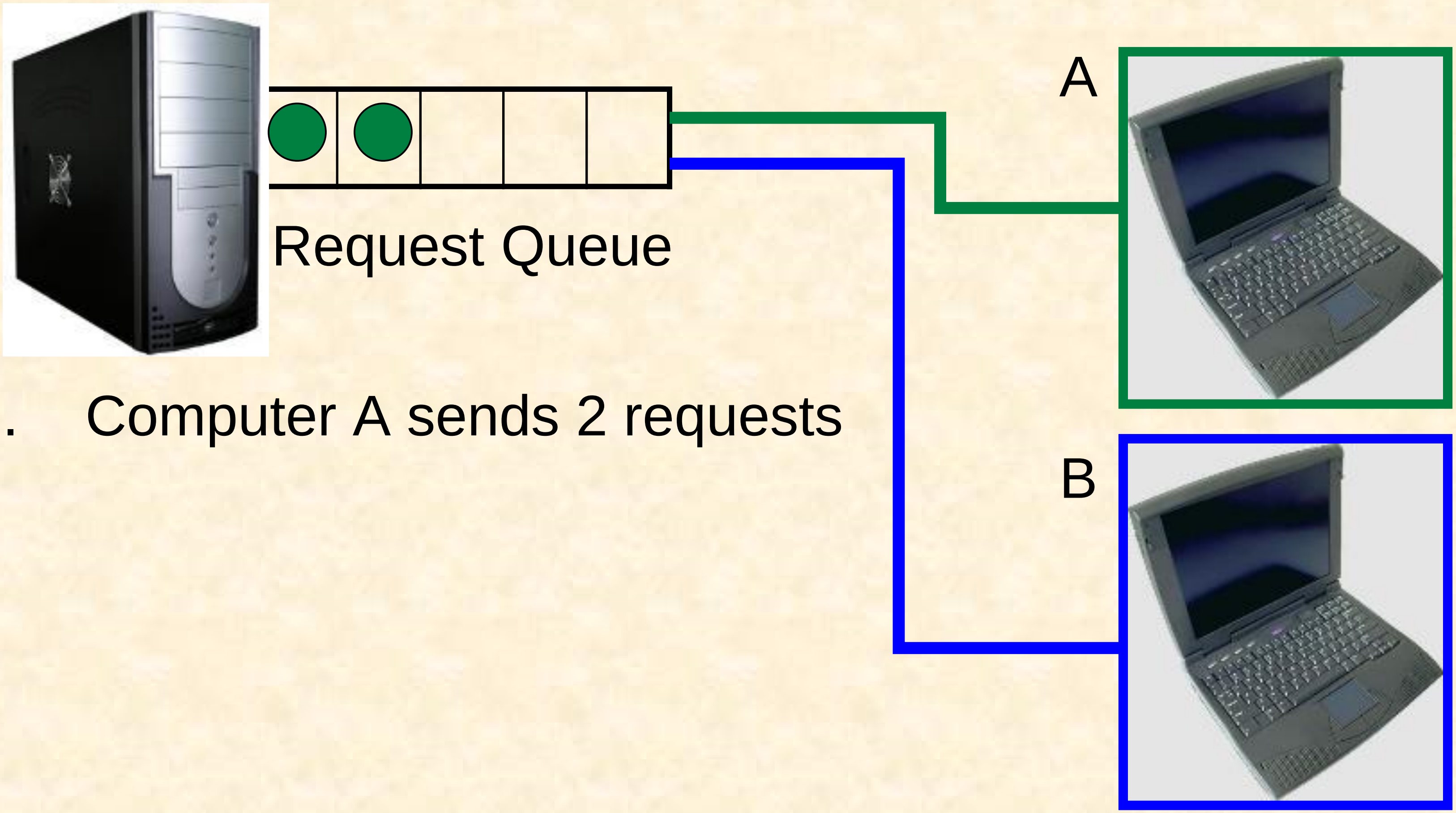
- Supports insertion/removal in FIFO order

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 the queue
`size()`	Number of elements in queue
`empty()`	Checks if queue has no elements

Queue Example: Web Browsing History

- The history starts empty
- **New pages are added to history on the “today” end**
- **Old pages are removed from history on the “30 days ago” end**
- *This particular kind of queue* (unlike STL queue) *allows iterating through elements*

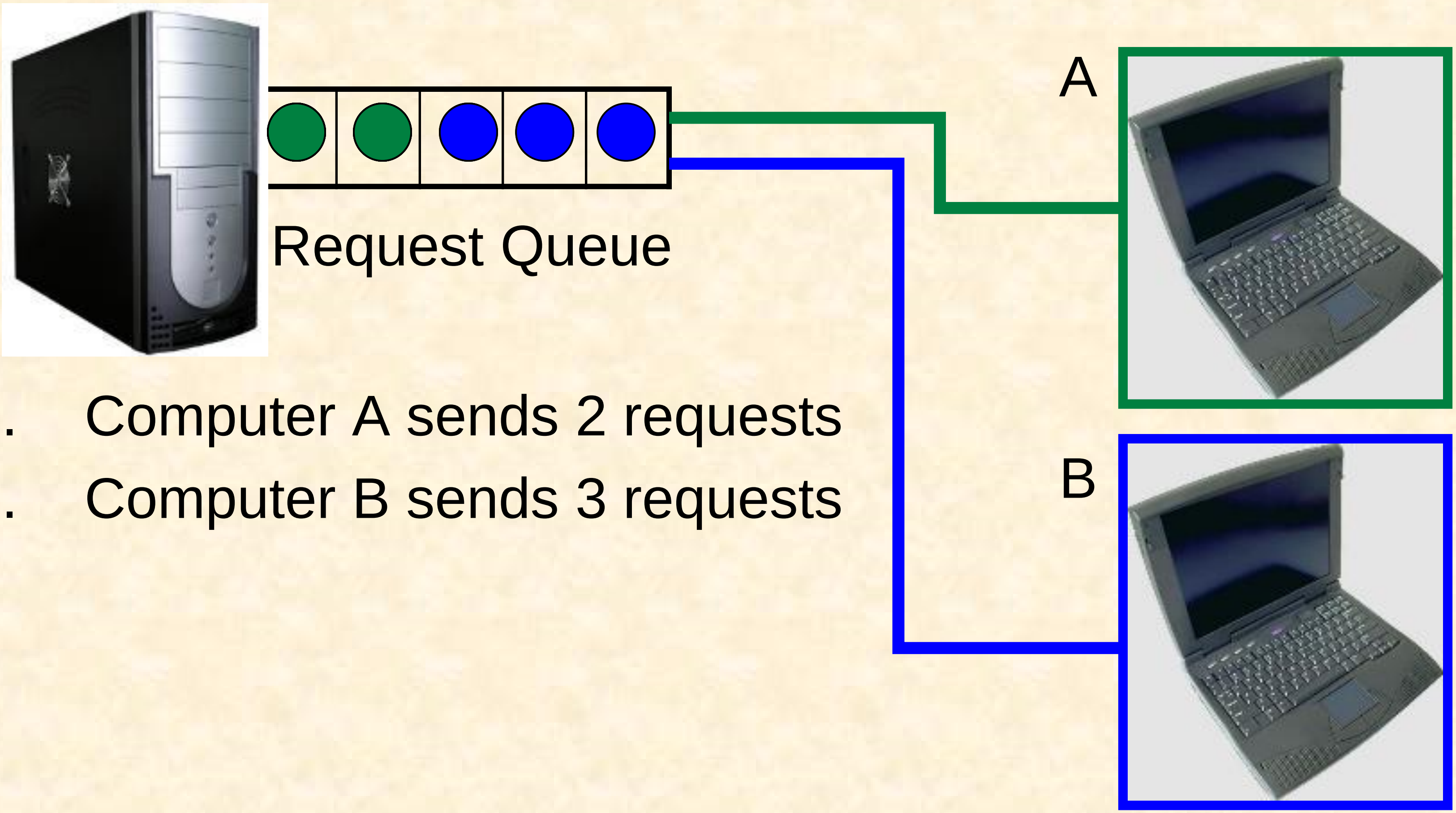
Queue Example: Request Queue of a Web Server



1. Computer A sends 2 requests

Should we use arrays or linked lists to implement this queue?

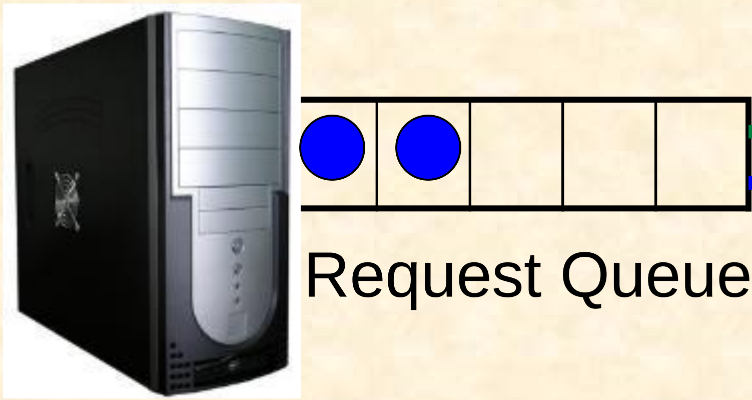
Queue Example:
Request Queue of a Web Server

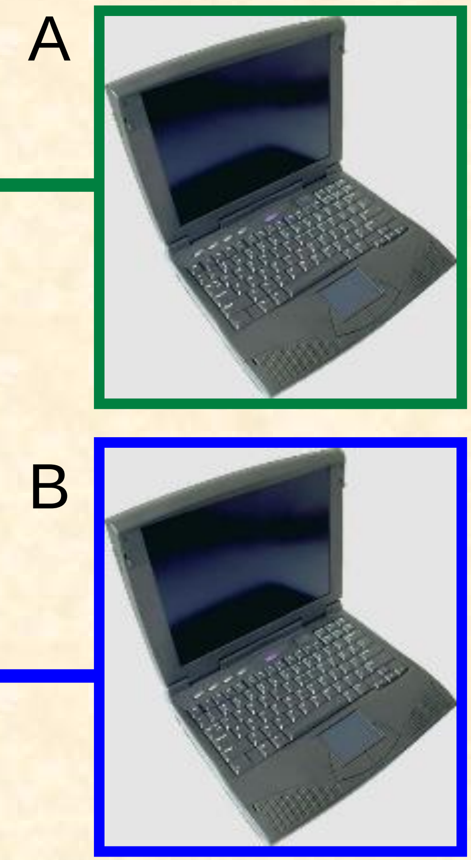


1. Computer A sends 2 requests
2. Computer B sends 3 requests

Should we use arrays or linked lists
to implement this queue?

Queue Example: Request Queue of a Web Server

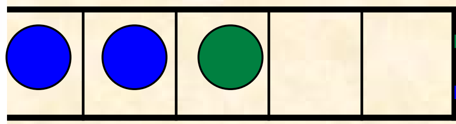


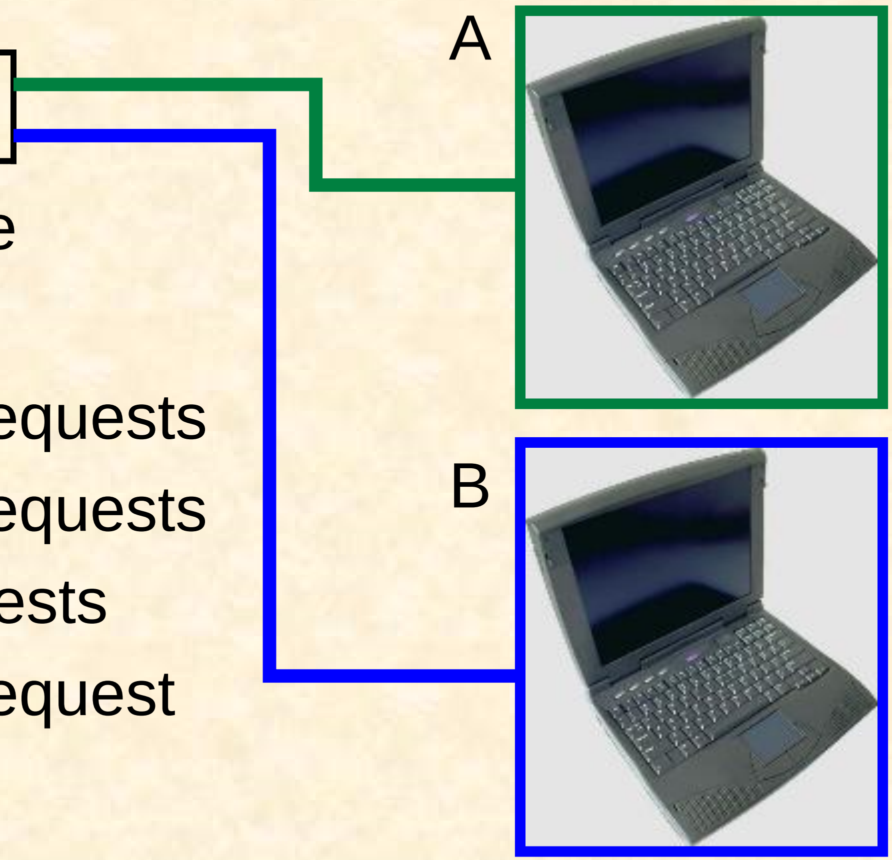


1. Computer A sends 2 requests
2. Computer B sends 3 requests
3. Server handles 3 requests

Should we use arrays or linked lists to implement this queue?

Queue Example:
Request Queue of a Web Server

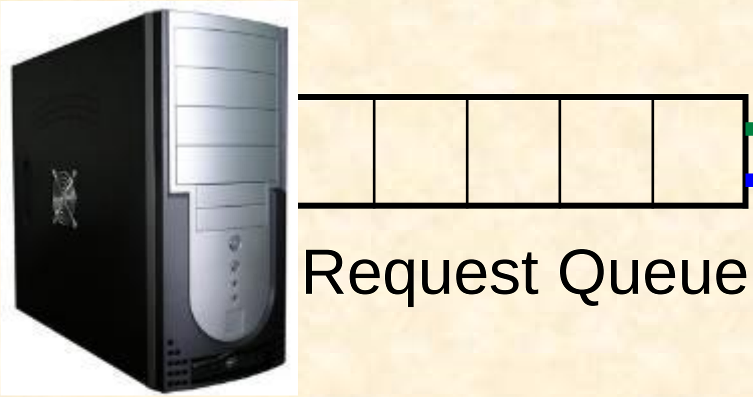


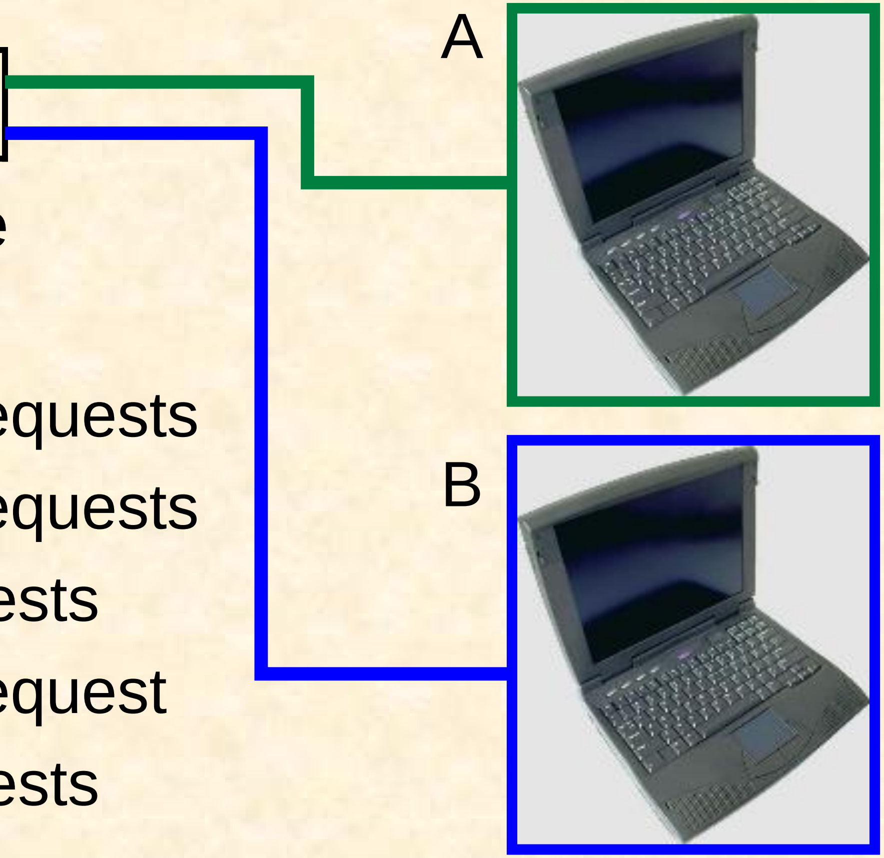


1. Computer A sends 2 requests
2. Computer B sends 3 requests
3. Server handles 3 requests
4. Computer A sends 1 request

Should we use arrays or linked lists
to implement this queue?

Queue Example: Request Queue of a Web Server



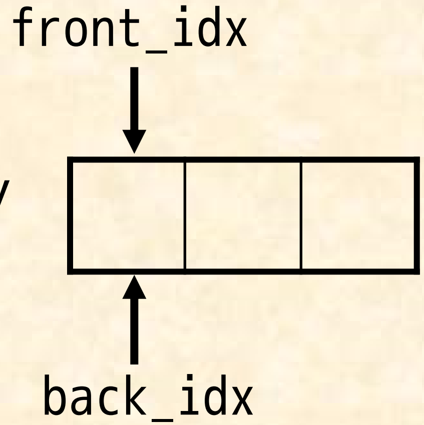


1. Computer A sends 2 requests
2. Computer B sends 3 requests
3. Server handles 3 requests
4. Computer A sends 1 request
5. Server handles 3 requests

Should we use arrays or linked lists to implement this queue?

Queues Using Arrays: Enqueue and Dequeue

Circular array of size 3

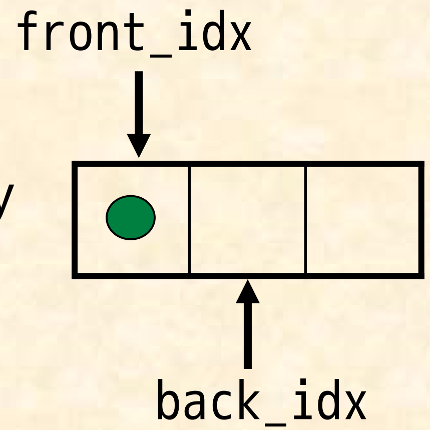


Event Sequence

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

Queues Using Arrays: Enqueue and Dequeue

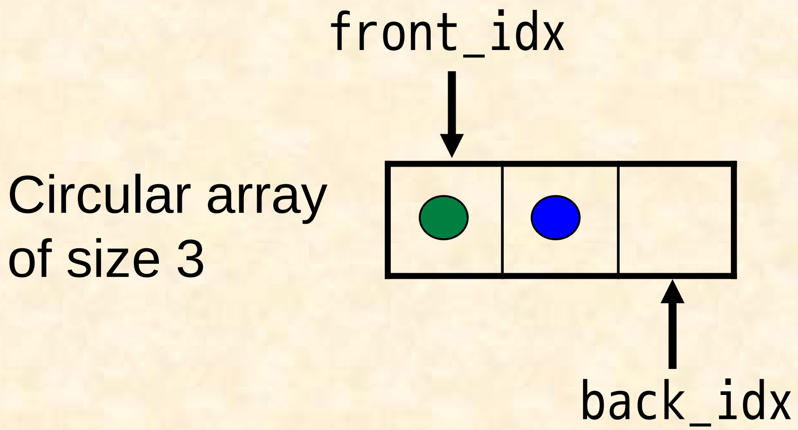
Circular array
of size 3



Event Sequence

1. `back_idx == front_idx` since array is empty
2. enqueue element

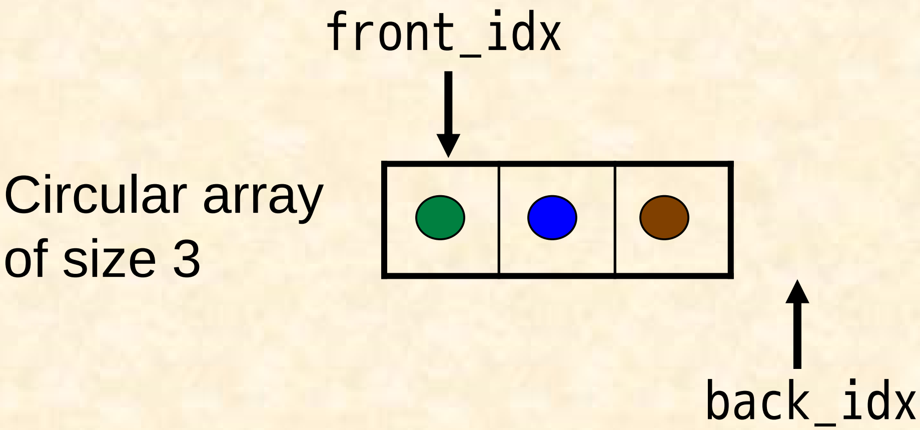
Queues Using Arrays: Enqueue and Dequeue



Event Sequence

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

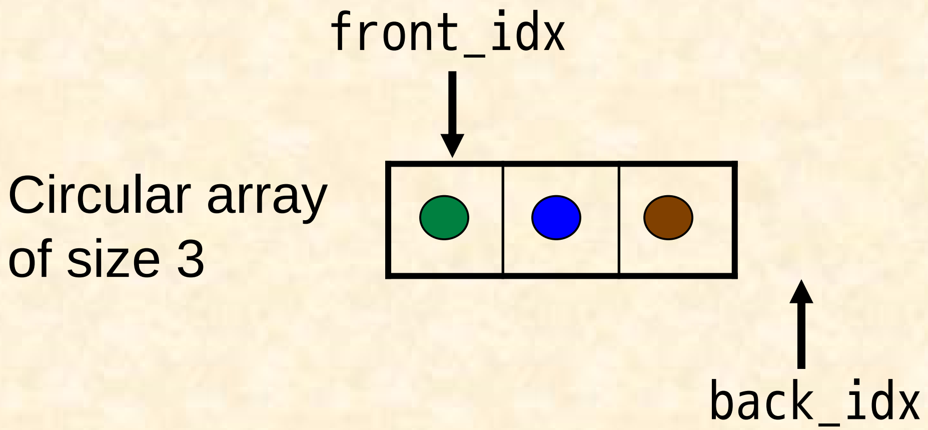
Queues Using Arrays: Enqueue and Dequeue

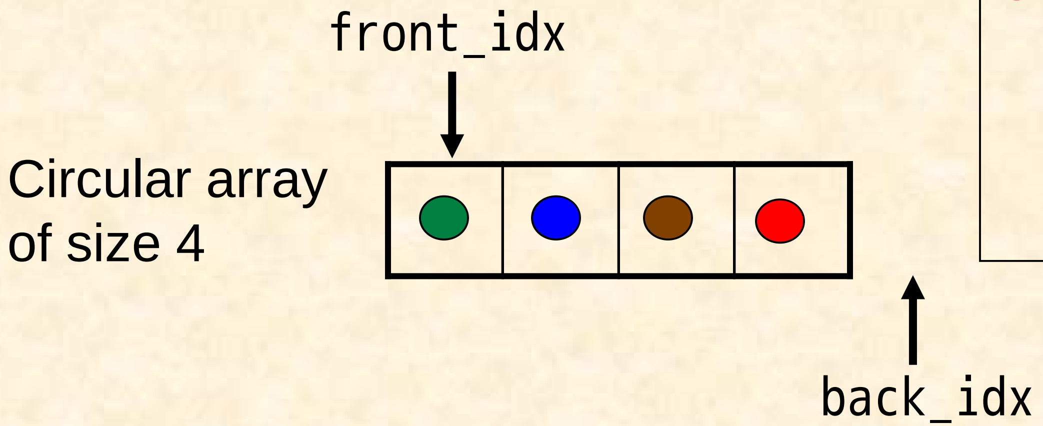


Event Sequence

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

Queues Using Arrays: Enqueue and Dequeue





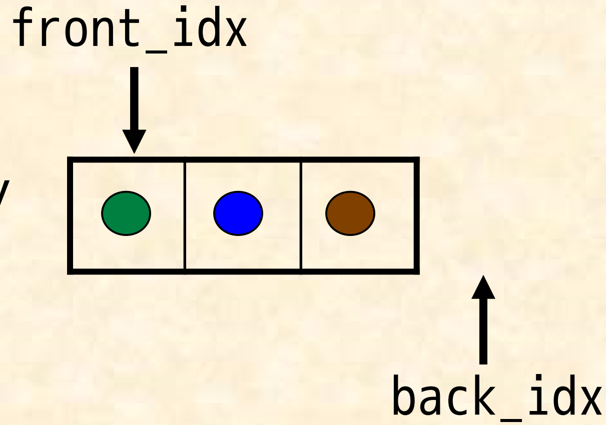
Event Sequence

1. `back_idx == front_idx` since array is empty
2. enqueue element
3. enqueue element
4. enqueue element
5. allocate more memory and enqueue element *

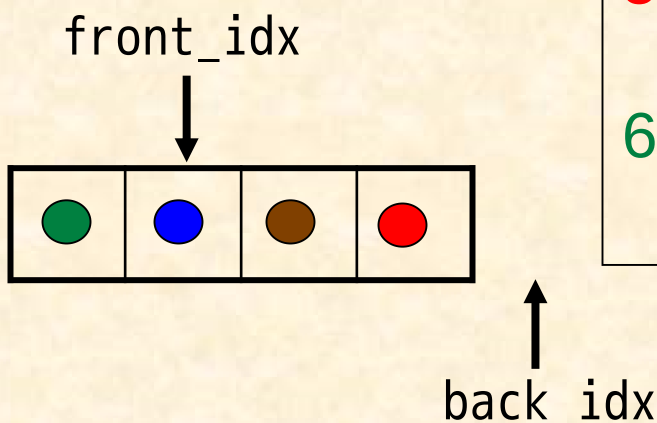
* When allocating more memory, it is more common to double memory

Queues Using Arrays: Enqueue and Deque

Circular array of size 3



Circular array of size 4



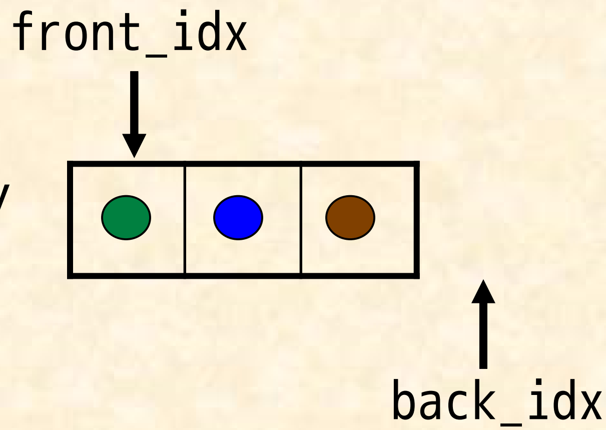
Event Sequence

1. `back_idx == front_idx` since array is empty
2. enqueue element
3. enqueue element
4. enqueue element
5. allocate more memory and enqueue element *
6. dequeue element

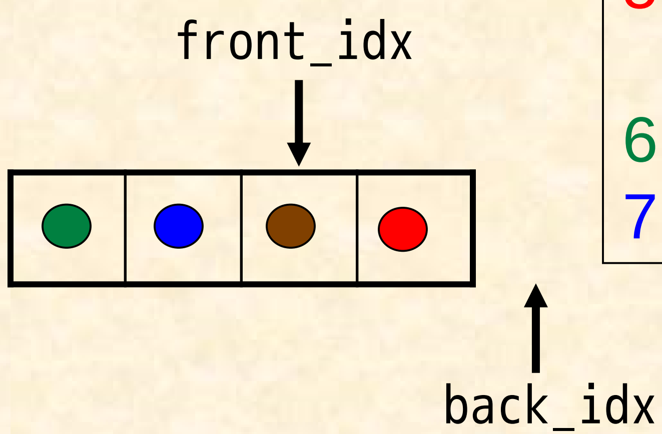
* When allocating more memory, it is more common to double memory

Queues Using Arrays: Enqueue and Dequeue

Circular array of size 3



Circular array of size 4



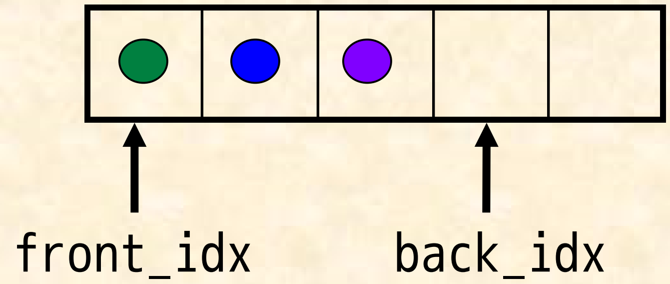
Event Sequence

1. `back_idx == front_idx` since array is empty
2. enqueue element
3. enqueue element
4. enqueue element
5. allocate more memory and enqueue element *
6. dequeue element
7. dequeue element

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

Queues Using Arrays

Use a circular array

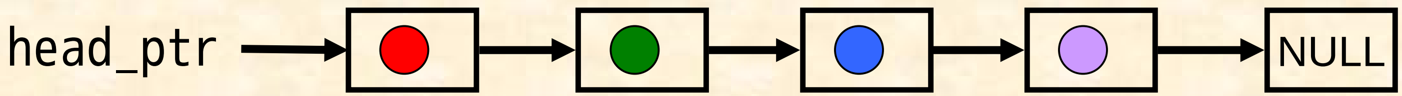


Method	Implementation
`enqueue(object)`	Increment `back_idx`, wrapping to front when end of allocated space is reached If `back_idx` becomes `front_idx`, reallocate array and unroll
`dequeue()`	Delete item at `front_idx` and increment `front_idx`
`object &front()`	Return reference to element at `front_idx`
`size()`	If (`back_idx >= front_idx`) returns `back_idx - front_idx` else returns `array_size + back_idx - front_idx`
`empty()`	returns `back_idx == front_idx`

What is the asymptotic runtime of each method?

Queues Using Linked Lists

Singly-linked is sufficient



Method	Implementation
`push(object)`	Append node to list
`pop()`	Delete head node of list
`object &front()`	Return reference to data in head node
`size()`	Use existing `LinkedList::size()` method Be careful: `size()` in STL `<slist>` takes $O(n)$ time (computes size from scratch every time)
`empty()`	Use existing `LinkedList::empty()` method

What is the asymptotic runtime of each method?

Is an array or linked list more efficient for queues?

Deque: a Queue and Stack in One

(Double-ended Queue)

- Pronounced “deck”
- 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*

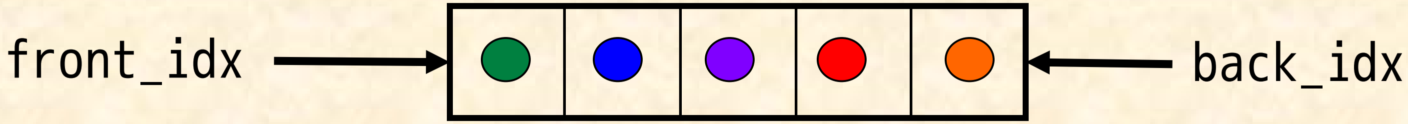
What’s another pseudo-word invented to name a data structure?

Trie (pronounced “try”) : a digital search tree

Deque Implementation

Circular Array

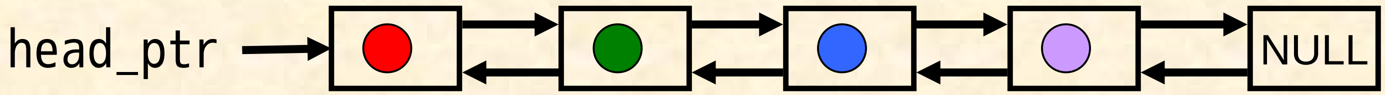
– `front_idx` and `back_idx` both get incremented/decremented



Doubly-linked list

– Singly-linked doesn't support efficient removal

– Other operations map directly to doubly-linked list operations



See details in STL header `<deque>` for another implementation

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
  - Largest determined by given ordering

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

What applications may benefit from a priority queue?

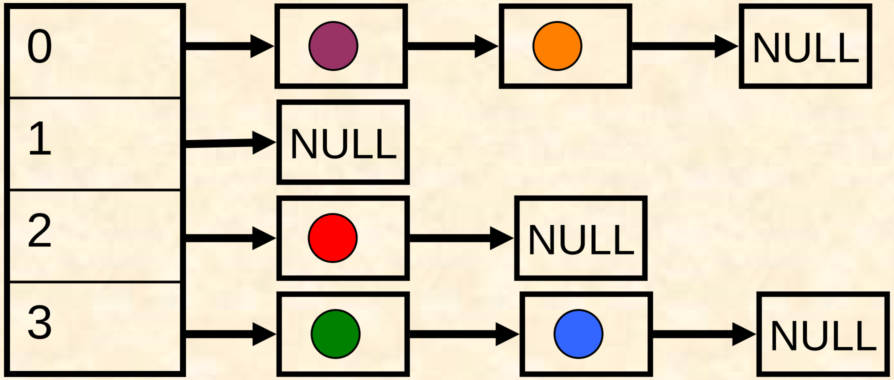
Priority Queue Implementation

STL maintains a heap on top of any random access container you choose

	Insert	Remove Max
Unsorted sequence container	$O(1)$	$O(n)$
Sorted sequence container	$O(\mathrm{n})$	$O(1)$
Heap (covered in future lecture)	$O(\log n)$	$O(\log n)$
Array of linked lists (for priorities of small integers)	$O(1)$	$O(1)$

Array of Linked Lists

Priority value used as index value in array

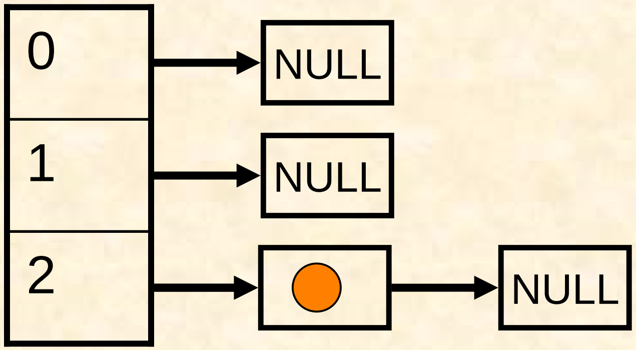


Priority Queue Example: Emergency Call Center

- 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

---

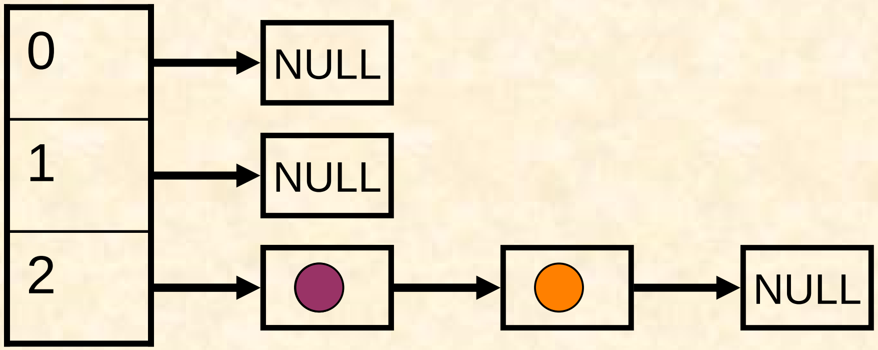
1. Level 2 call comes in



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

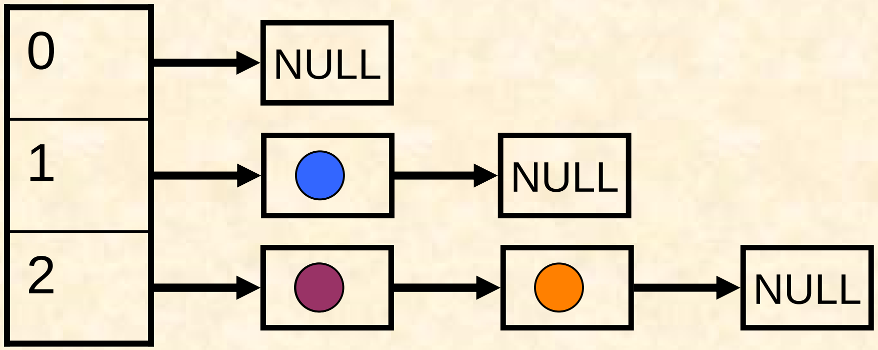
1. Level 2 call comes in
2. Level 2 call comes in



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

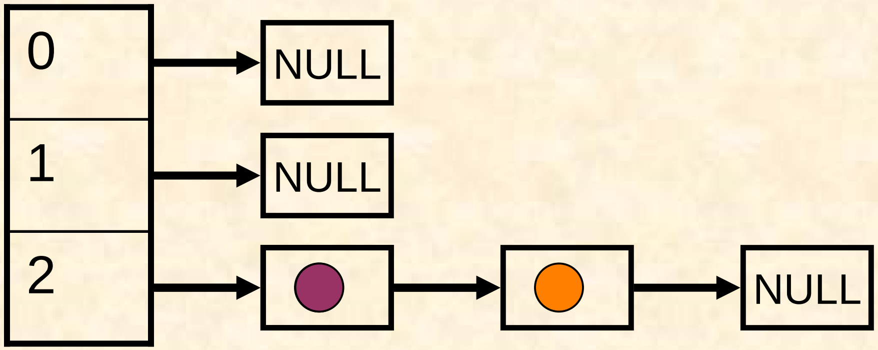
1. Level 2 call comes in
2. Level 2 call comes in
3. Level 1 call comes in



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

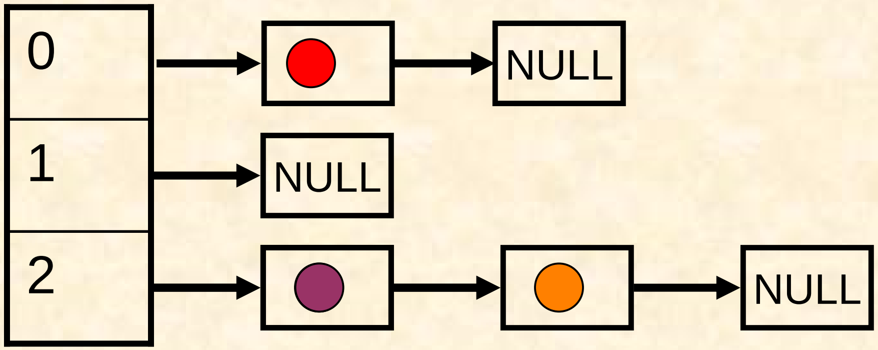


Priority Queue Example: Emergency Call Center

- 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

---

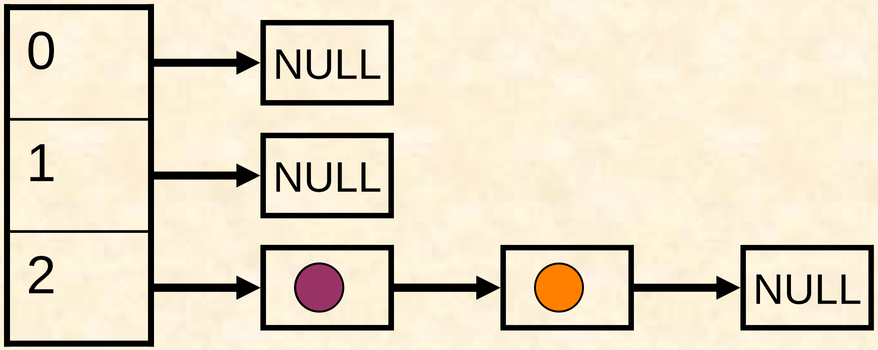
1. Level 2 call comes in
2. Level 2 call comes in
3. Level 1 call comes in
4. A call is dispatched
5. Level 0 call comes in



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



Stacks and Queues in STL

- 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	Queue
Default Underlying Container	`std::deque`	`std::deque`
Optional Underlying Container	`std::list` `std::vector`	`std::list`

Note: `std::list` is not the same as `std::slist`

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 (empirical data will be shown soon)
  - 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 Structure Engineering

- Exercise 1
  - Given a stack class (e.g., from STL)
  - Build a `MinStack` class with the same Big-O complexities, and an additional `getMin()` function that runs in $O(1)$ time
  - Note: all Big-O are worst-case
- Exercise 2
  - Same for a `MinQueue`

Algorithm Engineering: *Juggling with Stacks and Queues*

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

Implementation: Helper Function

```
template <typename T>
ostream &operator<<(ostream &out, const stack<T> &s) {
  // print the contents of a stack on a single line
  // e.g., cout << mystack << endl;
  stack<T> tmpStack = s; // deep copy
  while (!tmpStack.empty()) {
    out << tmpStack.top() << ' ';
    tmpStack.pop();
  } // while
  return out;
} // operator<<()
```

Implementation

```
template <typename T>
void genPerms(queue<T> &q, stack<T> &s) {
  // s: prefix of permutation, q: everything else
  unsigned size = q.size();
  if (q.empty()) {
    cout << s << '\n';
    return;
  } // if
  for (unsigned k = 0; k != size; k++) {
    s.push(q.front());
    q.pop();
    genPerms(q, s);
    q.push(s.top());
    s.pop();
  } // for
} // genPerms()
```

Better Helper Function

```
template <typename T>
ostream &operator<<(ostream &out, const vector<T> &s) {
  // print the contents of a vector on a single line
  // e.g., cout << myvector << endl;
  for (auto &el: s)
    out << el << ' ';

  return out;
} // operator<<()
```

Better Implementation

```
template <typename T>
void genPerms(deque<T> &q, vector<T> &s) {
  // s: prefix of permutation, q: everything else
  unsigned size = q.size();
  if (q.empty()) {
    cout << s << '\n';
    return;
  } // if
  for (unsigned k = 0; k != size; k++) {
    s.push_back(q.front());
    q.pop_front();
    genPerms(q, s);
    q.push_back(s.back());
    s.pop_back();
  } // for
} // genPerms()
```

Implementation: Sample Driver

```
int main() {
  unsigned n;
  string junk;
  cout << "Enter n: " << flush;
  while (!(cin >> n)) {
    cin.clear();
    getline(cin, junk);
    cout << "Enter n: " << flush;
  } // while

  vector<unsigned> s;
  deque<unsigned> q(n);
  iota(q.begin(), q.end(), 1);
  genPerms(q, s);
  return 0;
} // main()
```

Implement to Test

- **Q**: how does the recursive permutation enumerator compare to STL's function **`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
- Interview brainteaser
  - You are given four digits: 3 3 8 8 (can reorder)
  - Can use any combination of +, -, * and / (no power/exp, no concatenation)
  - Find a way to express **24**
  - Examples: 22=3+3+8+8, 23=(8-3)*3+8, 25=(8-3)*(8-3)

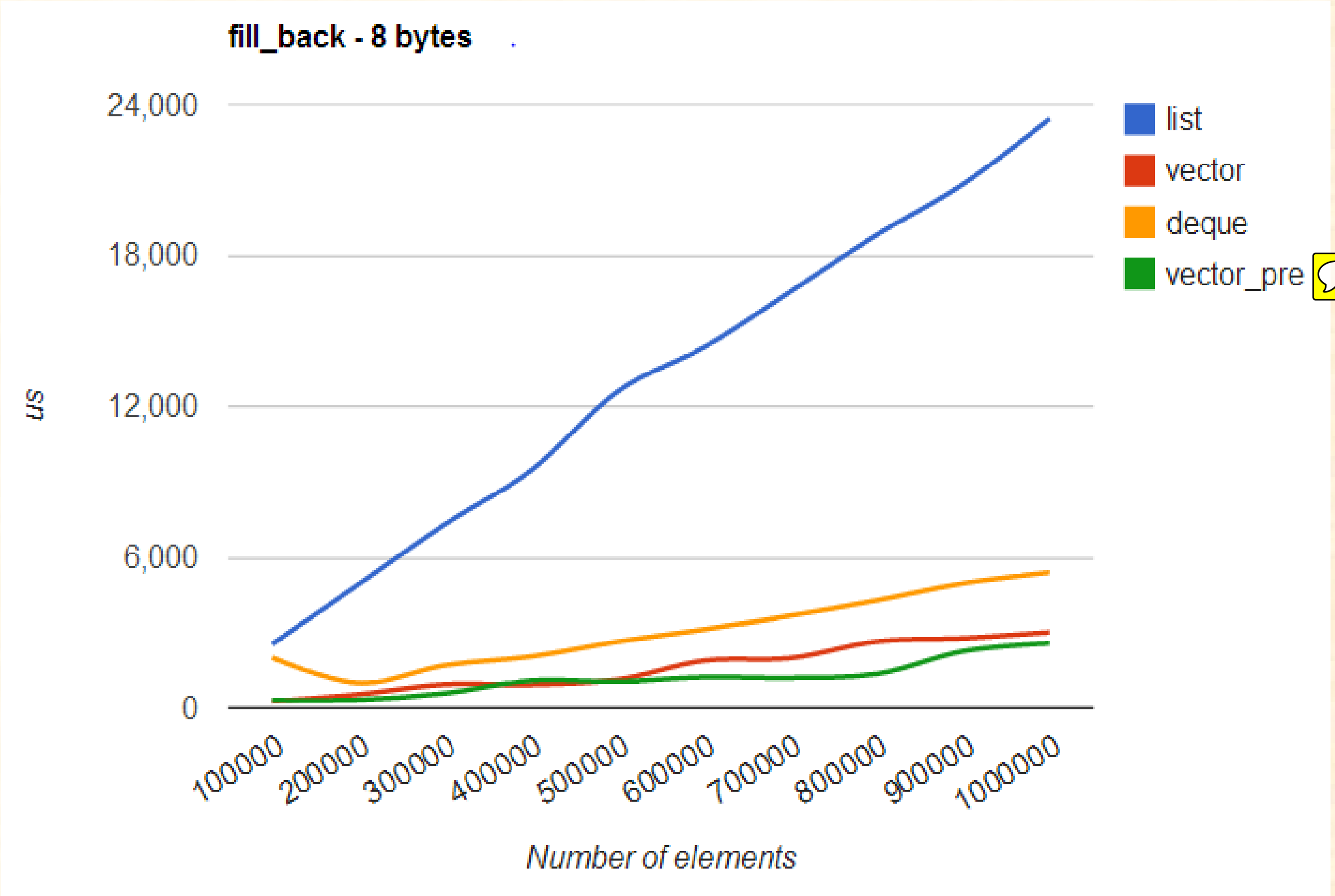
Relative Performance of STL Containers (1)

Filling an empty container with different values

vector_pre used `vector::resize()` (a single allocation)

Intel Core i7 Q820 @1.73GHz

GCC 4.7.2 (64b) -02 -std=c++11 -march=native

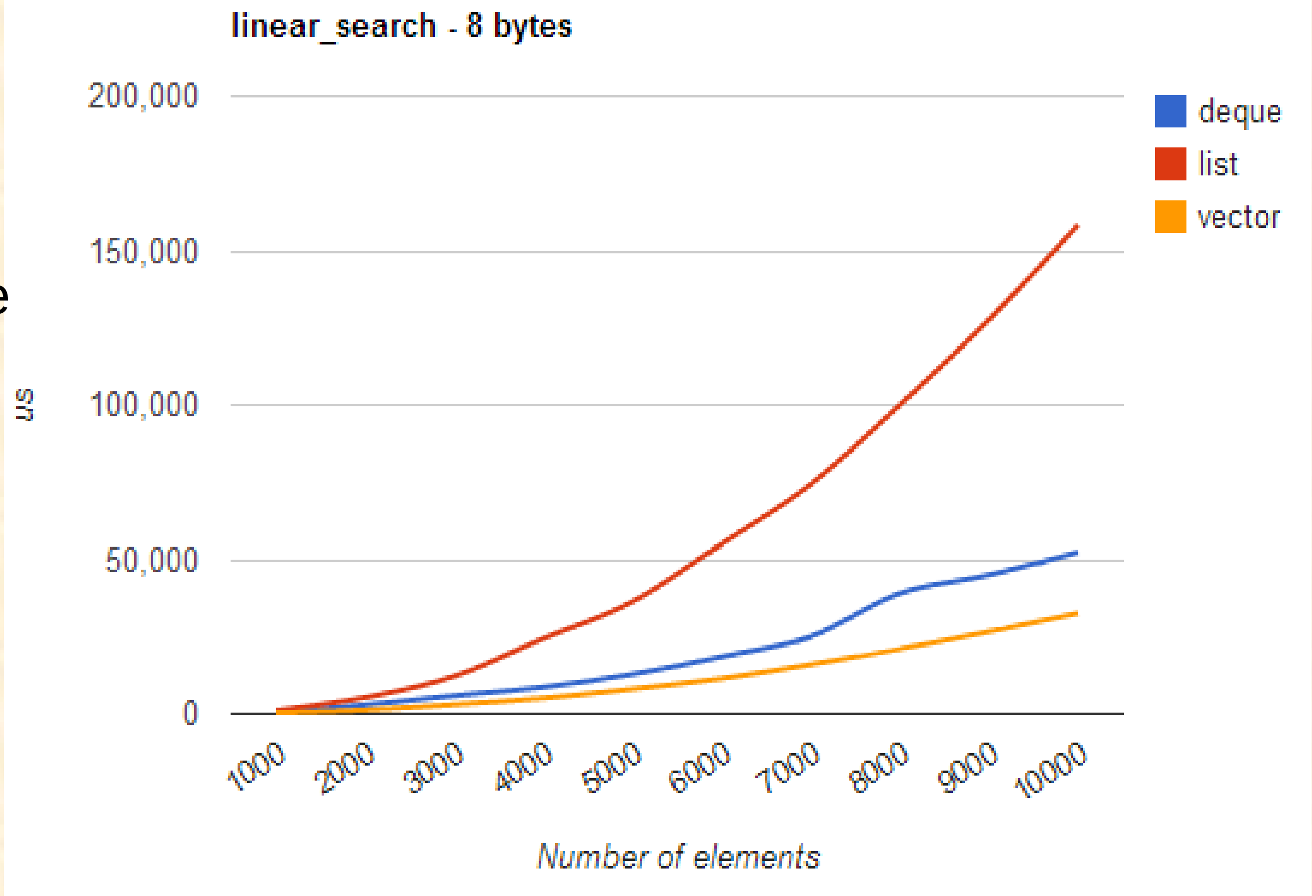


http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Relative Performance of STL Containers (2)

Fill the container with numbers [0, *N*], shuffle at random;

search for each value using `std::find()`



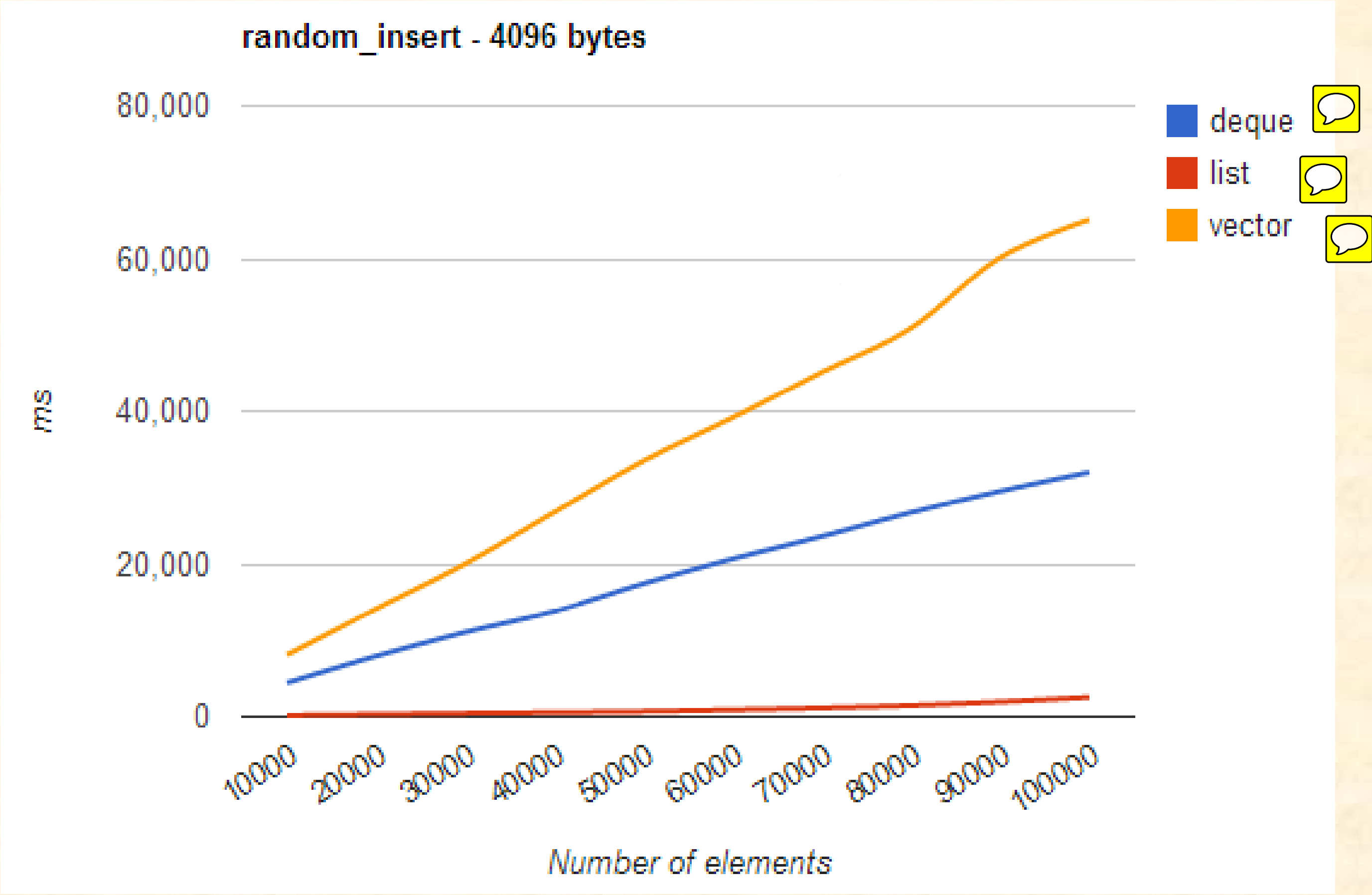
http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Relative Performance of STL Containers (3)

Fill the container with numbers [0, *N*], shuffle at random;

Pick a random position by linear search

Insert 1000 values



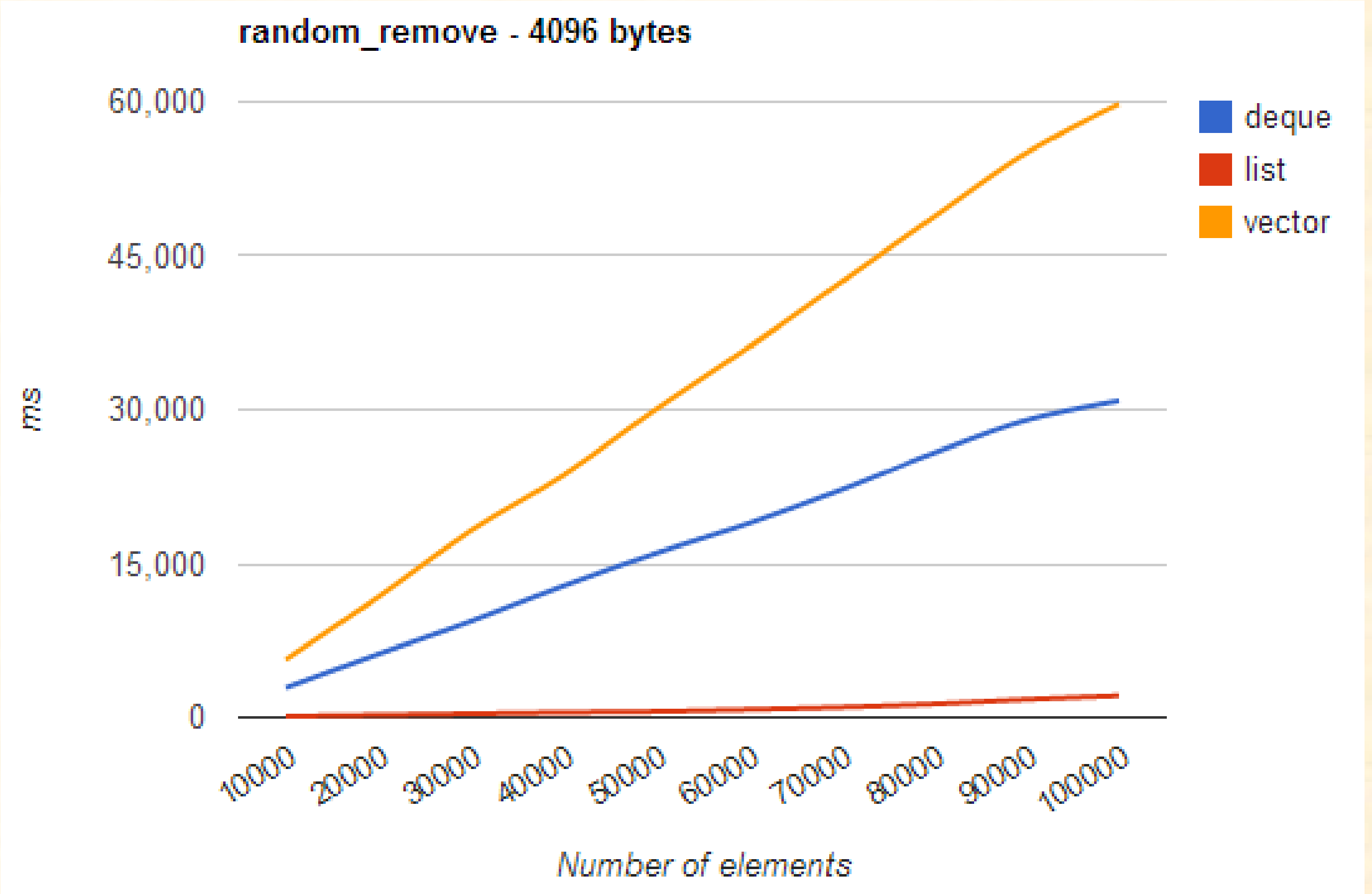
http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Relative Performance of STL Containers (4)

Fill the container with numbers [0, *N*], shuffle at random;

Pick a random position by linear search

Remove 1000 elements

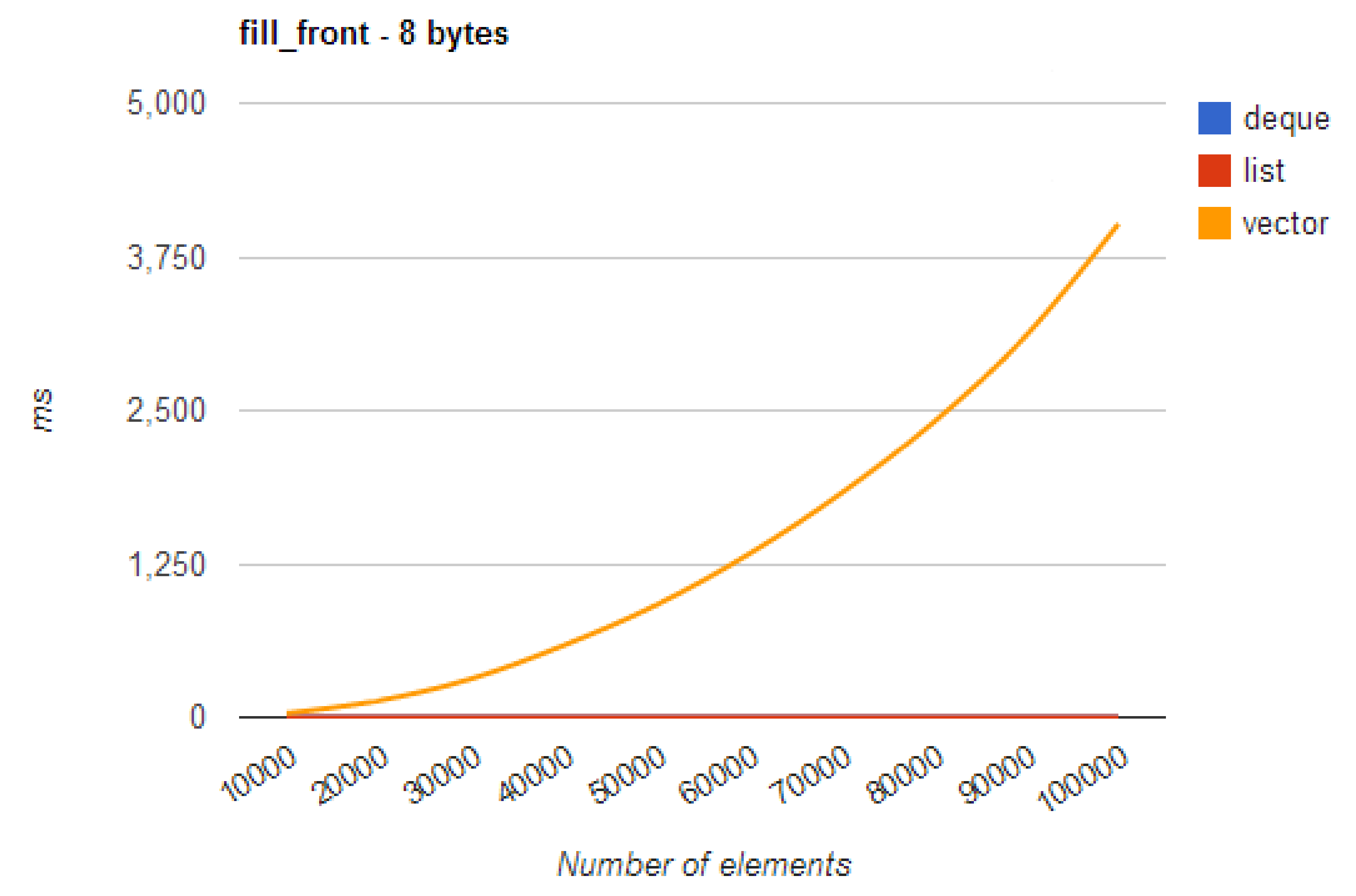


http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Relative Performance of STL Containers (5)

Insert new values at the front

A vector needs to move all prior elts, but a list does not



http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

What to study?

- What is an ADT?
- Define the following:
  - Stack
  - Queue
  - Deque
  - Priority queue

- How would you implement each ADT above?
- Compare the performance of vector, deque and list classes based on their implementation
- Describe several applications where one data structure would be more relevant than another