#20

investigating the efficiency of C++ containers

#cpp_friends, 14::30

Efficiency of standard containers

The standard library provides *containers* to store collections of same type.

The following containers are *dynamic*, meaning that they can grow or shrink as needed — their size is not fixed.

- **`std::list`**
- **`std::vector`**
- **`std::deque`**
- **`std::set`**
- **`std::unordered_set`**

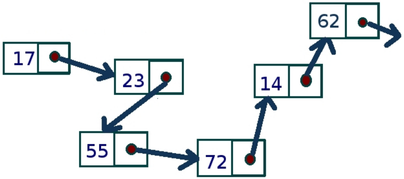
Efficiency of standard containers

- `std::list`
- `std::vector`
- `std::deque`
- `std::set`
- `std::unordered_set`

Do you know how do they *store the data internally*?

Can you tell which one is *the most optimal* for a given task?

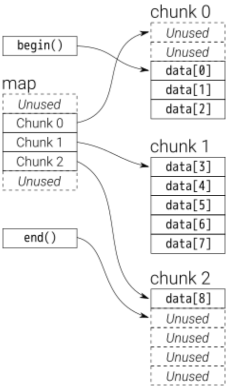
Bit of theory — `std::list`



Operation	Complexity
get(int index)	O(n)
push_back(T element) push_front(T element)	O(1)
insert(T element, int index)	O(1)
remove(int index)	O(1)

image source: a detail of https://commons.wikimedia.org/wiki/File:Linked_list_data_format.jpg

Bit of theory — `std::deque`



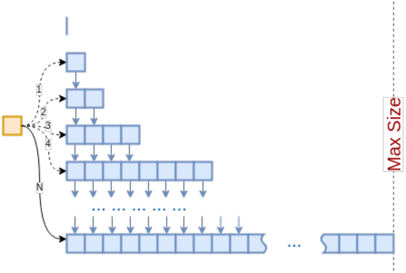
Operation	Complexity
get(int index)	O(1)
push_back(T element) push_front(T element)	O(1)
insert(T element, int index)	O(n)
remove(int index)	O(n)

O(1) insert and remove at both ends of deque

Note: data is not stored contiguously.

image source: part of https://stackoverflow.com/questions/6292332/what-really-is-a-deque-in-stl

Bit of theory — `std::vector`

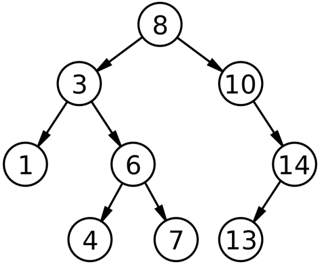


Operation	Complexity
get(int index)	O(1)
push_back(T element)	O(1), may be worse
push_front(T element)	O(n)
insert(T element, int index)	O(n)
remove_back	O(1)
remove(int index)	O(n)

Note: data is stored contiguously.

image source: part of https://stackoverflow.com/questions/52330010/what-does-stdvector-look-like-in-memory

Bit of theory — `std::set`

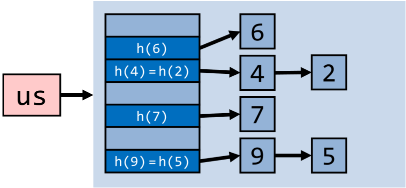


Operation	Complexity
get(int index)	*not provided*
find(T element)	O(log(n))
insert(T element)	O(log(n))
remove(T element)	O(log(n))

Note [1]: sequential access is not supported.
Note [2]: uses a *binary search tree*.

image source: part of https://en.wikipedia.org/wiki/Binary_search_tree

Bit of theory — `std::unordered_set`



Operation	Complexity
get(int index)	*not provided*
find(T element)	O(1) or O(n)
insert(T element)	O(1) or O(n)
remove(T element)	O(1) or O(n)

Note: all operation take average O(1) time.
It can go up to O(n) in worst case, but in general they perform well.
In more precise terms, unordered_set's operations are amortized O(1)

image source: https://hackingcpp.com/cpp/std/unordered_set_thumb.svg

Benchmark — storage of data

```cpp
template <typename T>
void test_storing() {
  std::cout << "Testing " << typeid(T).name() << " ... ";
  auto start {std::chrono::system_clock::now()};
  T container {};
  for (uint64_t i {0}; i < TEST_SIZE; ++i) {
    container.push_back(i);
  }
  auto finish {std::chrono::system_clock::now()};
  auto elapsed {std::chrono::duration_cast<std::chrono::milliseconds>(finish - start)};
  std::cout << "done, took " << elapsed.count() << " ms\n";
}
```

Trial	`vector`	`deque`	`list`	`set`	`unordered_set`
1.	70 ms	53 ms	309 ms	2745 ms	761 ms
2.	70 ms	54 ms	316 ms	2667 ms	730 ms
3.	71 ms	55 ms	332 ms	2700 ms	769 ms

Tested on an x86_64 machine with Ryzen 5600X processor. Full code: `storing.cpp`.

Benchmark — accessing data randomly

```
for (std::size_t k {0}; k < NUM_RANDOM_READS; k++) {
  read_value = *(container.cbegin() + INDEX_FIRST_QUARTER);
  read_value = *(container.cbegin() + INDEX_SECOND_QUARTER);
  read_value = *(container.cbegin() + INDEX_THIRD_QUARTER);
  read_value = *(container.cbegin() + INDEX_FOURTH_QUARTER);
}
```

Trial	`vector`	`deque`	`list`	`set`	`unordered_set`
1.	150 ms	304 ms	2449 ms	1743 ms	907 ms
2.	146 ms	295 ms	2437 ms	1754 ms	902 ms
3.	148 ms	308 ms	2446 ms	1729 ms	891 ms

Tested on an x86_64 machine with Ryzen 5600X processor. Full code: `reading_random.cpp`.

Benchmark — accessing data sequentially

```
for (std::size_t k {0}; k < NUM_SEQUENTIAL_READS; k++) {
  // begin s-th sequential read
  for (std::size_t r {0}; r < CONTAINER_SIZE; r++) {
    // read r-th element of the container (a deque or a vector)
    read_value = *(container.cbegin() + r);
  }
}
```

Trial	`vector`	`deque`	`list`	`set`	`unordered_set`
1.	11 ms	22 ms	1357 ms	143 ms	45 ms
2.	9 ms	21 ms	1353 ms	145 ms	45 ms
3.	10 ms	21 ms	1360 ms	143 ms	45 ms

Tested on an x86_64 machine with Ryzen 5600X processor. Full code: `reading_sequential.cpp`.

Benchmark — `std::vector` or `std::unordered_set`?

```cpp
void test_vector() {
  std::cout << "Testing std::vector ... " << std::flush;
  auto start {std::chrono::system_clock::now()};
  std::vector<tested_type> vec{};
  for (std::size_t i {0}; i < CONTAINER_SIZE; ++i) {
    vec.push_back(dist10G(rng));
  }
  std::sort(std::begin(vec), std::end(vec));
  auto finish {std::chrono::system_clock::now()};
  auto elapsed {std::chrono::duration_cast<timing_granularity>(finish - start)};
  std::cout << "done, took " << elapsed.count() << " units\n";
}
```

```cpp
void test_unordered_set() {
  std::cout << "Testing std::unordered_set ... " << std::flush;
  auto start {std::chrono::system_clock::now()};
  std::unordered_set<tested_type> myset{};
  for (std::size_t i {0}; i < CONTAINER_SIZE; ++i) {
    myset.insert(dist10G(rng));
  }
  auto finish {std::chrono::system_clock::now()};
  auto elapsed {std::chrono::duration_cast<timing_granularity>(finish - start)};
  std::cout << "done, took " << elapsed.count() << " units\n";
}

```

Container size	`vector`	`unordered_set`
10	15 μs	7 μs
100	23 μs	22 μs
1000	251 μs	241 μs
10K	3834 μs	2659 μs
1M	374 ms	426 ms
10M	4152 ms	6109 ms

Tested on an x86_64 machine with Ryzen 5600X processor. Full code: `vector_or_hashset.cpp`.

Why is that?

modern CPUs: every possible trick to boost performance

- accessing contiguous memory is *significantly* faster than accessing random addressses

- `std::list`: usually very fragmented (just a few insert/delete operations suffice)
- `std::deque`: fragmented chunks of contiguous memory
- `std::vector`: average theoretical performance, great practical performance
  - — especially for operations on the entire sequence
- `std::vector`: the only container that stores data *sequentially*

Key takeaways

- know the pros and cons of various standard containers
- know your machine too — caching, paging, speculative execution
  can provide a significant performance gain — or performance penalty
- linked list is still useful — e.g. great for caching data
- **`std::unordered_set`**: almost a silver bullet
- when in doubt — measure, then fix, then measure again
- **`std::vector`** is almost always good enough

Thank you!