BITE1523: COMPUTER GAME PROGRAMMING II

Chapter 4: Queue

Recap: Types of data structure: QUEUE

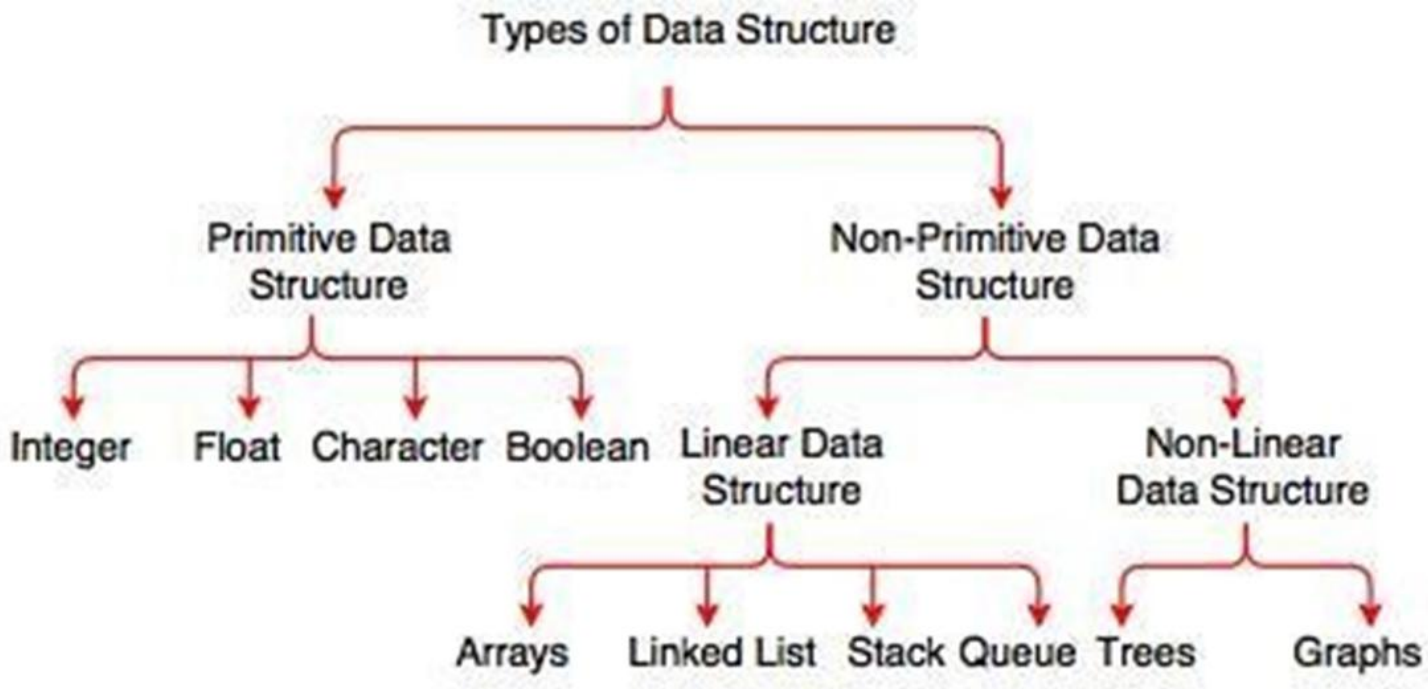


Fig. Types of Data Structure

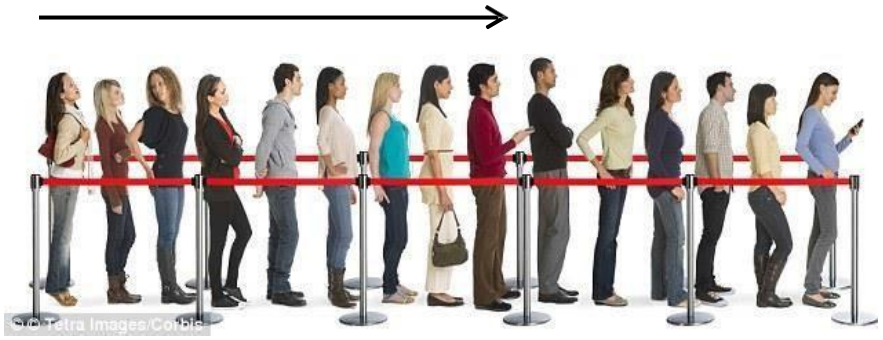
- **Objectives**
  - Understand queue concepts and applications.
  - Understand queue structure and operations that can be done on queue.
  - Understand and know how to implement queue using array and linked list : linear array, circular array, linear link list and circular list.

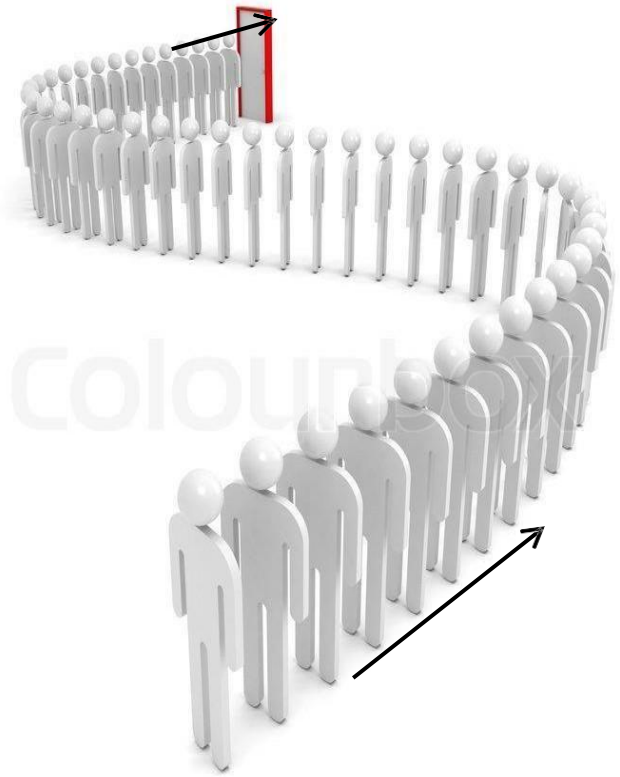
Queue ADT

A queue is an abstract data type where elements can be inserted (enqueue) and removed (dequeue) according to the following policies:

- Enqueue- inserts a new item in the back of the queue.
- Dequeue- removes the item at the front of the queue
- The above insert/remove policy is also called FIFO (First In-First Out).
- The first item inserted into a queue is the first item to leave.
- Queue is important in simulation & analyzing the behavior of complex systems

A queue is a data structure that is optimized for a specific access pattern: the “first in, first out” (FIFO) pattern that describes lines as we know them in everyday life.





When to use queues in C++

Queues are great at keeping track of tasks to process and are the right choice if you need to track the order in which items need to be processed.

When to avoid using queues in C++

We can't reference queue elements outside of the first and last ones, so if you're looking for access to, say, the sixth element of a 12-element list, a queue is not the right data structure for you. You might find an array better suited to this purpose.

In Short

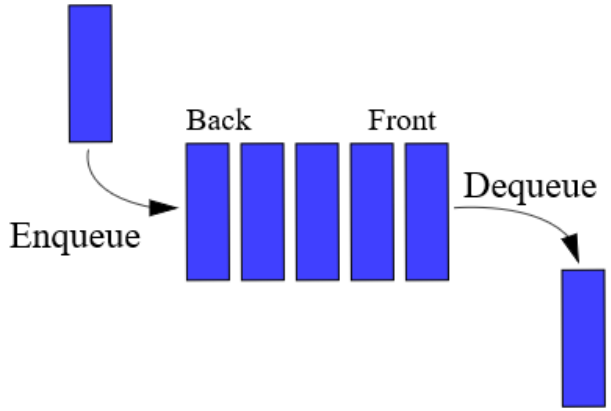
Use Queue When…	Don’t Use Queue When…
You need FIFO order	You need random access
BFS or Level Order Traversal	You need LIFO (use Stack)
Scheduling / Job Processing	You need sorting (use Priority Queue)
Thread-safe task handling	You need frequent mid-insert/delete

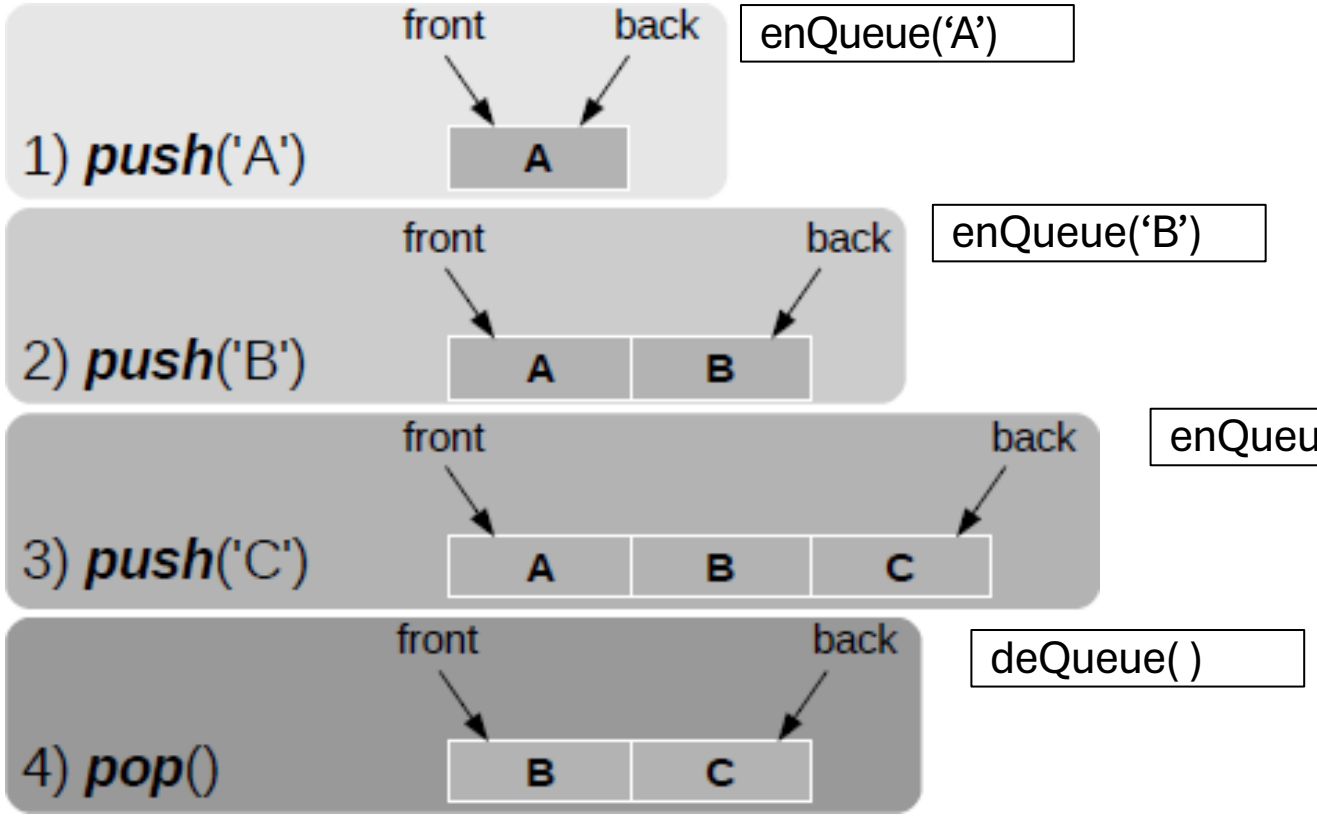
Queue Abstract Data Type

ADT queue operations

- Create an empty queue
- Destroy a queue
- Determine whether a queue is empty
- Add a new item to the queue
- Remove the item that was added earliest
- Retrieve at Front
- Retrieve at Back

Logical view of Queue





Queue Operations

enQueue(x)- places an item on the back of the queue.

deQueue()- removes the item at the front of the queue.

getFront()- access the item at the front of the queue.

getBack()- access the item at the back of the queue.

IsEmpty()– check whether the queue is empty

isFull()- check whether the queue is full

- There is three variables:
  - The size of the queue
  - Index to the beginning of the queue (front)
  - Index to the end of the queue (back)

Queue Implementation, cont..

Array-based

- Linear
- Circular

Pointer-based : Linked list

- Linear
- Circular

std::queue class in C++

Functions	Description
front()	Access the front element of the queue.
back()	Access the end element of the queue.
empty()	Check whether a queue is empty or not.
size()	Returns the number of elements in the queue.
push()	Adding an element at the back of the queue.
push_range()	Adding multiple elements at the end of queue.
emplace()	Add the constructs element in top of the queue.
pop()	Delete the front element of the queue.
swap()	Swap two queues.

Create C++ STL Queue

In order to create a queue in C++, we first need to include the queue header file.

```
#include <queue>
```

Once we import this file, we can create a queue using the following syntax:

```
queue<type> q;
```

Here, type indicates the data type we want to store in the queue. For example,

```
// create a queue of integer data type
queue<int> integer_queue;

// create a queue of string data type
queue<string> string_queue;
```

Push Element

```
#include <iostream>
#include <queue>

using namespace std;

int main() {

    // create a queue of string
    queue<string> animals;

    // push elements into the queue
    animals.push("Cat");
    animals.push("Dog");

    cout << "Queue: ";

    // print elements of queue
    // loop until queue is empty
    while (!animals.empty()) {

        // print the element
        cout << animals.front() << ", ";

        // pop element from the queue
        animals.pop();
    }

    cout << endl;

    return 0;
}
```

In the example, we have created a queue of strings called animals. Here, we have used the push() method to add elements to the end of the queue.

```
// push elements into the queue
   animals.push("Cat");
   animals.push("Dog");
```

Push Element

```cpp
#include <iostream>
#include <queue>

using namespace std;

int main() {

    // create a queue of string
    queue<string> animals;

    // push elements into the queue
    animals.push("Cat");
    animals.push("Dog");

    cout << "Queue: ";

    // print elements of queue
    // loop until queue is empty
    while (!animals.empty()) {

        // print the element
        cout << animals.front() << ", ";

        // pop element from the queue
        animals.pop();
    }

    cout << endl;

    return 0;
}
```

Instead of directly printing the contents of the queue, we have used a while loop and various queue methods.

```cpp
while (!animals.empty()) {

        // print the element
        cout << animals.front() << ", ";

        // pop element from the queue
        animals.pop();
    }
```

Push Element

```cpp
#include <iostream>
#include <queue>

using namespace std;

int main() {

    // create a queue of string
    queue<string> animals;

    // push elements into the queue
    animals.push("Cat");
    animals.push("Dog");

    cout << "Queue: ";

    // print elements of queue
    // loop until queue is empty
    while (!animals.empty()) {

        // print the element
        cout << animals.front() << ", ";

        // pop element from the queue
        animals.pop();
    }

    cout << endl;

    return 0;
}
```

This is because the STL queue is an STL Container Adapter that provides restrictive access to make it behave like a standard queue data structure.

As a result, we cannot iterate through the queue like iterating through vectors or other containers.

Instead, we print its front and then pop the element repeatedly inside a loop until the queue is empty.

Remove Element from a Queue

We can use the pop() method to remove an element from the front of a queue.

Pop Element

```
#include <iostream>
#include <queue>
using namespace std;

// function prototype for
display_queue utility
void display_queue(queue<string> q);

int main() {

    // create a queue of string
    queue<string> animals;

    // push element into the queue
    animals.push("Cat");
    animals.push("Dog");
    animals.push("Fox");

    cout << "Initial Queue: ";
    display_queue(animals);

    // remove element from queue
    animals.pop();

    cout << "Final Queue: ";
    display_queue(animals);

    return 0;
}
```

```
// utility function to display queue
void display_queue(queue<string> q) {
    while (!q.empty()) {
        cout << q.front() << ", ";
        q.pop();
    }

    cout << endl;
}
```

In the example, we have used the pop() method to remove an element from the queue.

Initially, the contents of the queue are "Cat", "Dog" and "Fox".

```
// remove element from queue
    animals.pop();
```

Pop Element

```cpp
#include <iostream>
#include <queue>
using namespace std;

// function prototype for
display_queue utility
void display_queue(queue<string> q);

int main() {

    // create a queue of string
    queue<string> animals;

    // push element into the queue
    animals.push("Cat");
    animals.push("Dog");
    animals.push("Fox");

    cout << "Initial Queue: ";
    display_queue(animals);

    // remove element from queue
    animals.pop();

    cout << "Final Queue: ";
    display_queue(animals);

    return 0;
}
```

```cpp
// utility function to display queue
void display_queue(queue<string> q) {
    while (!q.empty()) {
        cout << q.front() << ", ";
        q.pop();
    }

    cout << endl;
}
```

Here animals.pop() removes the element at the front of the queue i.e. the element inserted at the very beginning which is "Cat".

Hence, the final queue contains the elements "Dog" and "Fox".

Access Element from a Queue

We can access the elements of a queue using the following methods:

front() - returns the element from the front of the queue
back() - returns the element from the back of the queue

Access Element

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {

// create a queue of int
queue<int> nums;

// push element into the queue
nums.push(1);
nums.push(2);
nums.push(3);

// get the element at the front
int front = nums.front();
cout << "First element: " << front
<< endl;

// get the element at the back
int back = nums.back();
cout << "Last element: " << back
<< endl;

return 0;
}
```

In the above example, we have used the front() and back() methods to get the first and last elements from a queue of integers called nums.

We can get the first element i.e. the element at the front using:

```cpp
// returns 1
nums.front();
```

Here, 1 was inserted first so it is at the front.

Access Element

```
#include <iostream>
#include <queue>
using namespace std;

int main() {

// create a queue of int
queue<int> nums;

// push element into the queue
nums.push(1);
nums.push(2);
nums.push(3);

// get the element at the front
int front = nums.front();
cout << "First element: " << front
<< endl;

// get the element at the back
int back = nums.back();
cout << "Last element: " << back
<< endl;

return 0;
}
```

Similarly, we find the last element i.e. the rear (back) element using:

```
// returns 3
nums.back();
```

Here, 3 was inserted last, so it is at the back.

Get the size of a Queue

We use the size() method to get the number of elements in the queue

Get the size of a Queue

```
#include <iostream>
#include <queue>
using namespace std;

int main() {

// create a queue of string
queue<string> languages;

// push element into the queue
languages.push("Python");
languages.push("C++");
languages.push("Java");

// get the size of the queue
int size = languages.size();
cout << "Size of the queue: " <<
size;

return 0;
}
```

In the above example, we have created a queue of strings called languages and added three elements to it.

Then, we used the size() method to find the number of elements in the queue:

```
// returns 3
languages.size();
```

Since we have added three elements to the queue, `languages.size();` returns 3.

Check if the Queue is Empty

We use the empty() method to check if the queue is empty. This method returns:

1 (true) - if the queue is empty
0 (false) - if the queue is not empty

Queue if Empty

```
#include <iostream>
#include <queue>
using namespace std;

int main() {

    // create a queue of string
    queue<string> languages;

    cout << "Is the queue empty? ";

    // check if the queue is empty
    if (languages.empty()) {
        cout << "Yes" << endl;
    }
    else {
        cout << "No" << endl;
    }

    cout << "Pushing elements..." <<
endl;

// push element into the queue
    languages.push("Python");
    languages.push("C++");

    cout << "Is the queue empty? ";

    // check if the queue is empty
    if (languages.empty()) {
        cout << "Yes";
    }
    else {
        cout << "No";
    }

    return 0;
}
```

In the above example, we have used the empty() method to determine if the queue is empty,

```
// check if the queue is empty
   if (languages.empty()) {
       cout << "Yes" << endl;
   }
   else {
       cout << "No" << endl;
   }
```

Initially, the queue has no elements in it. So languages.empty() returns true.

We then add elements to the queue.
Then we again use languages.empty() again to determine if the queue is empty. This time, it returns false.

Create C++ STL Deque

In order to create a deque in C++, we first need to include the deque header file.

```
#include <deque>
```

Once we import this file, we can create a deque using the following syntax :

```
deque<type> dq;
```

Here, type indicates the data type we want to store in the deque.For example,

```
// create a deque of integer data type
deque<int> dq_integer;

// create a deque of string data type
deque<string> dq_string;
```

Apa beza Queue and Deque?

🔄 Queue (Barisan Biasa)

- FIFO – First In, First Out
- Masuk dari satu hujung (belakang) dan keluar dari satu hujung (depan)
  - Contoh:Macam beratur beli makanan – siapa sampai dulu, dia dapat dulu.
- Operasi utama:
  - enqueue(item) – masuk belakang
  - dequeue() – keluar depan

🔁 Deque (Double Ended Queue)

- Boleh masuk dan keluar dari kedua-dua hujung!
- Gabungan antara Queue dan Stack
  - Contoh:Macam laluan dua hala — boleh masuk/keluar dari depan atau belakang, ikut keperluan.
- Operasi utama:
  - addFront(item) – masuk depan
  - addRear(item) – masuk belakang
  - removeFront() – keluar depan
  - removeRear() – keluar belakang

Initialize Deque

We can initialize a C++ deque in the following ways:

// method 1: initializer list

- deque<int> deque1 = {1, 2, 3, 4, 5};

// method 2: uniform initialization

- deque<int> deque2 {1, 2, 3, 4, 5};

Here, both deque1 and deque2 are initialized with values 1, 2, 3, 4, 5.

✅ Method 1: Initializer List

```
deque<int> deque1 = {1, 2, 3, 4, 5};
```

- Ini guna copy-list initialization.
- Diperkenalkan dalam C++11.
- Compiler akan guna std::initializer_list untuk mengisi deque.
- Ada tanda =.

✅ Method 2: Uniform Initialization

```
deque<int> deque2 {1, 2, 3, 4, 5};
```

- Ini guna direct-list initialization, juga dari C++11.
- Lebih "modern" dan digalakkan dalam style C++ baru.
- Tak ada =.
- Compiler akan cuba guna constructor deque(initializer_list<int>) terus.

Example: C++ Deque

```
#include <iostream>
#include <deque>
using namespace std;

// function prototype
void display_deque(deque<int>);

int main() {

    // uniform initialization
    deque<int> deque1{ 1, 2, 3, 4, 5 };

    cout << "deque1 = ";

    // display elements of deque1
    for (int num : deque1) {
        cout << num << ", ";
    }

    return 0;
}
```

In the above example, we have declared and initialized a deque named deque1 using uniform initialization.

```
deque<int> deque1{ 1, 2, 3, 4, 5 };
```

Then, we have displayed the deque contents using a ranged for loop.

```
for (int num : deque1) {
        cout << num << ", ";
   }
```

#2 to initialize Deque - Fill Constructor Method

Using the Fill Constructor Method, we can initialize a deque by specifying its size and element. For example,

```
// fill constructor
deque<int> deque1(5, 12);
```

Here, we have initialized a deque of size 5 with values 12.

This is equivalent to

```
deque<int> deque1 = { 12, 12, 12, 12, 12 };
```

Notice that all the elements have the same value.

We can also initialize a deque by copying elements from another deque. This is known as the range method.

#3 to initialize Deque - Create a deque from another deque.

We can copy one deque to another using the range method initialization.

```
deque<int> deque1 = { 1, 2, 3, 4, 5 };

// copy all elements of deque1 to deque2
deque<int> deque2(deque1.begin(), deque1.end());
```

Here, we have first created a deque named deque1.
Then, we created another deque called deque2 by copying all the elements of deque1.

We can also specify the range of elements we want to copy.For example,

```
// copy elements from index 1 to index 2 of deque1
deque<int> deque3(deque1.begin() + 1, deque1.begin() + 3);
```

Here, the start point is deque1.begin() + 1, which points to index 1 of deque1.
The end point is deque1.begin() + 3, which points to index 3.

However, deque3 is going to be{ 2, 3 } because the end point is exclusive i.e.the end point will not be copied.

C++ Deque Methods

Methods	Description
push_back()	inserts element at the back
push_front()	inserts element at the front
pop_back()	removes element from the back
pop_front()	removes element from the front
front()	returns the element at the front
back()	returns the element at the back
at()	sets/returns the element at specified index
size()	returns the number of elements
empty()	returns true if the deque is empty

Insert Elements to a Deque

We can use the following methods to insert elements:

push_back() - insert elements at the end
push_front() - insert elements at the beginning

Insert Elements to a Deque

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {

    deque<int> nums{ 2, 3 };

    cout << "Initial Deque: ";
    for (const int& num : nums) {
        cout << num << ", ";
    }

    // add integer 4 at the back
    nums.push_back(4);

    // add integer 1 at the front
    nums.push_front(1);

    cout << "\nFinal Deque: ";
    for (const int& num : nums) {
        cout << num << ", ";
    }

    return 0;
}
```

In the example, we have initialized a deque of integers named nums with the elements 2 and 3.

Then, we inserted the integer 4 to the back of the nums.

```
// nums = {2, 3, 4}
nums.push_back(4);
```

Insert Elements to a Deque

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {

    deque<int> nums{ 2, 3 };

    cout << "Initial Deque: ";
    for (const int& num : nums) {
        cout << num << ", ";
    }

    // add integer 4 at the back
    nums.push_back(4);

    // add integer 1 at the front
    nums.push_front(1);

    cout << "\nFinal Deque: ";
    for (const int& num : nums) {
        cout << num << ", ";
    }

    return 0;
}
```

Finally, we inserted 1 at the front of the deque.

// nums = {1, 2, 3, 4}
nums.push_front(1);

Note: We can also use the insert() and emplace() methods to add elements to the deque.

Access Elements from a Deque

We can use the following methods to access elements of the deque:

front() - returns element at the front
back() - returns element at the back
at() - returns element at the specified index

Access Elements from a Deque

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3 };

cout << "Front element: " <<
nums.front() << endl;
cout << "Back element: " <<
nums.back() << endl;
cout << "Element at index 1: " <<
nums.at(1) << endl;
cout << "Element at index 0: " <<
nums[0];

return 0;
}
```

In the example,

nums.front() - returns the element at the front of the deque i.e. 1
nums.back() - returns the element at the back of the deque i.e. 3
nums.at(1) - returns the element at index 1 i.e. 2
nums[0] - returns the element at index 0 i.e. 1
Note: While we can use the [] operator to access deque elements, it is preferable to use the at() method.

This is because at() throws an exception whenever the deque is out of bounds, while [] gives a garbage value.

Change Elements of a Deque

We can use the at() method to change the element of the deque.

Change Elements of a Deque

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {

    deque<int> nums = { 1, 2 };

    cout << "Initial Deque: ";

    for (const int& num : nums) {
        cout << num << ", ";
    }

    // change elements at the
index
    nums.at(0) = 3;
    nums.at(1) = 4;

    cout << "\nUpdated Deque: ";

    for (const int& num : nums) {
        cout << num << ", ";
    }

    return 0;
}
```

In the example, the initial contents of the nums deque are {1, 2}.

Then, we have used the at() method to change the elements at the specified indexes:

nums.at(0) = 2; - changes the value at index 0 changes to 3
nums.at(1) = 4; - changes the value at index 1 to 4

Remove Elements from a Deque

We can remove the elements of a deque using the following methods:

pop_back() - removes the element from the back
pop_front() - removes the element from the front

Remove Elements from a Deque

```cpp
#include <iostream>
#include <deque>
using namespace std;

// function prototype for display_deque()
void display_deque(deque<int>);

int main() {
    deque<int> nums{ 1, 2, 3 };

    cout << "Initial Deque: ";
    display_deque(nums);

    // remove element from the back
    nums.pop_back();

    cout << "\nDeque after pop_back(): ";
    display_deque(nums);

// remove element from the front
    nums.pop_front();

    cout << "\nDeque after pop_front(): ";
    display_deque(nums);

    return 0;
}
```

```cpp
// utility function to print deque elements
void display_deque(deque<int> dq) {
    for (const int& num : dq) {
        cout << num << ", ";
    }
}
```

In the above example, we have initialized an integer deque named nums with the elements {1, 2, 3}.

Then, we used pop_back() and pop_front() to remove elements from nums.

nums.pop_back(); // removes 3
names.pop_front(); // removes 1

So the final deque becomes {2}.

C++ Deque Iterators

Iterators can be used to point to the memory address of a deque element.

Nice, jom kita sembang santai pasal **C++ Deque Iterators** dengan analogi yang senang faham.

🔁 Apa Itu Iterator?

- Sebelum kita masuk bab deque Iterator, kita kena faham dulu apa itu iterator secara umum.
- Analogi: ***Anggap kau ada barisan kerusi dalam satu dewan. Setiap kerusi ada nombor. Iterator ni macam penunjuk atau orang yang berdiri di salah satu kerusi***, dan dia boleh:
  - Tunjuk kerusi sekarang **(*it)**
  - Gerak ke kerusi seterusnya **(it++)**
  - Gerak ke belakang **(it--)**
  - Lompat ke kerusi ke-n **(it + n, it - n)**

🎯 Kenapa Kena Guna Iterator?

- Kalau kau nak **loop** atau **akses elemen dalam container** (macam deque), iterator bagi cara yang **seragam** dan **efisien** tak kira apa jenis container (vector, deque, list, dll).

Jum tengok contoh coding

```cpp
#include <deque>
#include <iostream>
using namespace std;

int main() {
    deque<int> dq = { 10, 20, 30, 40, 50 };
    deque<int>::iterator it;

    for (it = dq.begin(); it != dq.end(); ++it) {
        cout << *it << " ";
    }
    return 0;
}
```

🔍 Apa Yang Berlaku Dalam Kod Tu?

- Analogi Deque: *Bayangkan deque sebagai kereta api yang ada gerabak depan & belakang, dan iterator ni macam pejalan kaki yang jalan dari gerabak pertama ke akhir sambil tengok nombor kat setiap gerabak.*

🔍 Apa Yang Berlaku Dalam Kod Tu?

- ***dq.begin()*** *= pejalan kaki mula dari gerabak pertama.*
- ***dq.end()*** *= bukan gerabak terakhir, tapi satu tempat selepas gerabak terakhir (macam pagar penamat).*
- ****it*** *= tengok nombor atas gerabak sekarang.*
- ***++it*** *= jalan ke gerabak seterusnya.*

💡 Jenis Iterator Dalam Deque:

Jenis Iterator	Fungsi	Analogi
begin()	Mula dari depan	Mula jalan dari gerabak depan
end()	Penamat (beyond last)	Dah lepas gerabak akhir
rbegin()	Mula dari belakang	Jalan dari gerabak belakang ke depan
rend()	Penamat reverse	Habis jalan ke depan (dalam reverse)
cbegin()	Const iterator depan	Tengok je, tak boleh ubah gerabak
crbegin()	Const reverse iterator	Tengok dari belakang, tak boleh ubah

Initialize Deque

We can create a deque iterator using the following syntax :

```
deque<type>::iterator iterator_name;
```

For example,

```
// iterator for int deque
deque<int>::iterator iter_int;

// iterator for double deque
deque<double>::iterator iter_double;
```

Access Elements Using Deque Iterators

We can access the elements in deque using iterators.

Remove Elements from a Deque

```
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3 };

// declare an iterator to deque
deque<int>::iterator dq_iter;

// make iterator point to first element
dq_iter = nums.begin();

// print value pointed by itertor using *
int first_element = *dq_iter;

cout << "nums[0] = " << first_element << endl;

// make iterator point to element at index 1
dq_iter = nums.begin() + 1;
int element_index1 = *dq_iter;

cout << "nums[1] = " << element_index1 << endl;

// make iterator point to last element
dq_iter = nums.end() - 1;
int last_element = *dq_iter;

cout << "nums[2] = " << last_element;

return 0;
}
```

In the above example, we have used iterators to access elements in the deque. Initially, we created an iterator that can point to a deque of integers:

```
// declare an iterator to deque
deque<int>::iterator dq_iter;
```

Remove Elements from a Deque

```
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3 };

// declare an iterator to deque
deque<int>::iterator dq_iter;

// make iterator point to first element
dq_iter = nums.begin();

// print value pointed by itertor using *
int first_element = *dq_iter;

cout << "nums[0] = " << first_element << endl;

// make iterator point to element at index 1
dq_iter = nums.begin() + 1;
int element_index1 = *dq_iter;

cout << "nums[1] = " << element_index1 << endl;

// make iterator point to last element
dq_iter = nums.end() - 1;
int last_element = *dq_iter;

cout << "nums[2] = " << last_element;

return 0;
}
```

Then, we have used the dq_iter iterator to point to the following elements:

1. The First Element

```
// make iterator point to first element
dq_iter = nums.begin();
```

Here, the begin() method returns an iterator that points to the first element.

Remove Elements from a Deque

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3 };

// declare an iterator to deque
deque<int>::iterator dq_iter;

// make iterator point to first element
dq_iter = nums.begin();

// print value pointed by itertor using *
int first_element = *dq_iter;

cout << "nums[0] = " << first_element << endl;

// make iterator point to element at index 1
dq_iter = nums.begin() + 1;
int element_index1 = *dq_iter;

cout << "nums[1] = " << element_index1 << endl;

// make iterator point to last element
dq_iter = nums.end() - 1;
int last_element = *dq_iter;

cout << "nums[2] = " << last_element;

return 0;
}
```

2. Element at Index 1

```cpp
dq_iter = nums.begin() + 1;
```

The code begin() + 1 returns an iterator that points to the element at index 1.

Note: To generalize, `nums.begin() + 1` points to the element at index i.

Remove Elements from a Deque

```
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3 };

// declare an iterator to deque
deque<int>::iterator dq_iter;

// make iterator point to first element
dq_iter = nums.begin();

// print value pointed by itertor using *
int first_element = *dq_iter;

cout << "nums[0] = " << first_element << endl;

// make iterator point to element at index 1
dq_iter = nums.begin() + 1;
int element_index1 = *dq_iter;

cout << "nums[1] = " << element_index1 << endl;

// make iterator point to last element
dq_iter = nums.end() - 1;
int last_element = *dq_iter;

cout << "nums[2] = " << last_element;

return 0;
}
```

3. The Last Element

```
// make iterator point to last element
dq_iter = nums.end() - 1;
```

Notice that we are using `nums.end() - 1` instead of `nums.end()`.

This is because the end() method iterator points to the iterator past the last element. So, to get the final element, we are subtracting 1.

Remove Elements from a Deque

```
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3 };

// declare an iterator to deque
deque<int>::iterator dq_iter;

// make iterator point to first element
dq_iter = nums.begin();

// print value pointed by itertor using *
int first_element = *dq_iter;

cout << "nums[0] = " << first_element << endl;

// make iterator point to element at index 1
dq_iter = nums.begin() + 1;
int element_index1 = *dq_iter;

cout << "nums[1] = " << element_index1 << endl;

// make iterator point to last element
dq_iter = nums.end() - 1;
int last_element = *dq_iter;

cout << "nums[2] = " << last_element;

return 0;
}
```

4. Get the Element Value

After using `dq_iter` to point to a certain element, we use the indirection operator * to get the value of that element:

```
// point to the first element
dq_iter = nums.begin()

// returns 1
int first_element = *dq_iter;
```

Here, `*dq_iter` gives the value of the element pointed to by the `dq_iter` iterator.

Exercise 1

How to remove elements at the specified index?

We can remove the element at the specified index using the erase() method. For example,

```
deque<int> nums{ 1, 2, 3, 4, 5 };

// removes element at index 1
// nums becomes {1, 3, 4, 5}
nums.erase(nums.begin() + 1);
```

The code above removes the element at index 1 from the nums deque.

So, we can delete the element at any index using:

```
nums.erase(nums.begin() + index);
```

```cpp
#include <iostream>
#include <deque>
using namespace std;

// utility function to print deque elements
void display_deque(deque<int> dq) {
for (const int& num : dq) {
cout << num << ", " <<endl;
}
}

int main() {

deque<int> nums{ 1, 2, 3 };

// declare an iterator to deque
deque<int>::iterator dq_iter;

// make iterator point to first element
dq_iter = nums.begin();

// print value pointed by itertor using *
int first_element = *dq_iter;

cout << "nums[0] = " << first_element << endl;

// make iterator point to element at index 1
dq_iter = nums.begin() + 1;
int element_index1 = *dq_iter;

cout << "nums[1] = " << element_index1 << endl;

// make iterator point to last element
dq_iter = nums.end() - 1;
int last_element = *dq_iter;

cout << "nums[2] = " << last_element << endl;

cout << "Before Remove at Index 1 " << endl;

display_deque(nums);

cout << "After Remove at Index 1 " << endl;

// removes element at index 1
// nums becomes {1, 3, 4, 5}
nums.erase(nums.begin() + 1);

display_deque(nums);

return 0;
}
```

Exercise 2

How to remove elements in a certain index range?

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3, 4, 5 };

// removes elements from index 1 to 2
nums.erase(nums.begin() + 1, nums.begin() + 3);

for (auto num : nums) {
cout << num << ", ";
}

return 0;
}
```

Initially, nums contains the elements {1, 2, 3, 4, 5}.

Then we remove the elements from index 1 to index 2 using,

```cpp
// removes elements from index 1 to 2
nums.erase(nums.begin() + 1, nums.begin() + 3);
```

Notice that the end point of the range is nums.begin() + 3, but the element at index 3 is not removed.

This is because the end point of the erase() function is exclusive.

🧪 Ringkasan FOR LOOP

Gaya For Loop	Cara
Traditional loop	for (int i = 0; i < nums.size(); i++)
Iterator loop	for (auto it = nums.begin(); it != nums.end(); ++it)
Range-based loop	for (auto num : nums)

Exercise 3

How to remove all elements of the deque?

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3, 4, 5 };

// removes elements from index 1 to 2
nums.erase(nums.begin() + 1, nums.begin() + 3);

for (auto num : nums) {
cout << num << ", ";
}

cout << endl;
cout << "size :";
cout << nums.size() << endl;

nums.clear();

cout << "size :";
cout << nums.size() << endl;
return 0;
}
```

Just use

```cpp
nums.clear();
```

Exercise 4

How to determine the size of the deque?

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main() {

deque<int> nums{ 1, 2, 3, 4, 5 };

// removes elements from index 1 to 2
nums.erase(nums.begin() + 1, nums.begin() + 3);

for (auto num : nums) {
cout << num << ", ";
}

cout << endl;
cout << "size :";
cout << nums.size() << endl;


}
```

We can use the size() method to find the number of elements in the deque.

```
deque<int> nums {1, 2};

// returns 2
cout << nums.size();
```

We can also determine if the deque is empty using the empty() method:

```
// returns false
cout << nums.empty();
```

The End

End of Queue
‘Topic