ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chương 7:
Ứng dụng các cấu trúc dữ liệu cơ bản

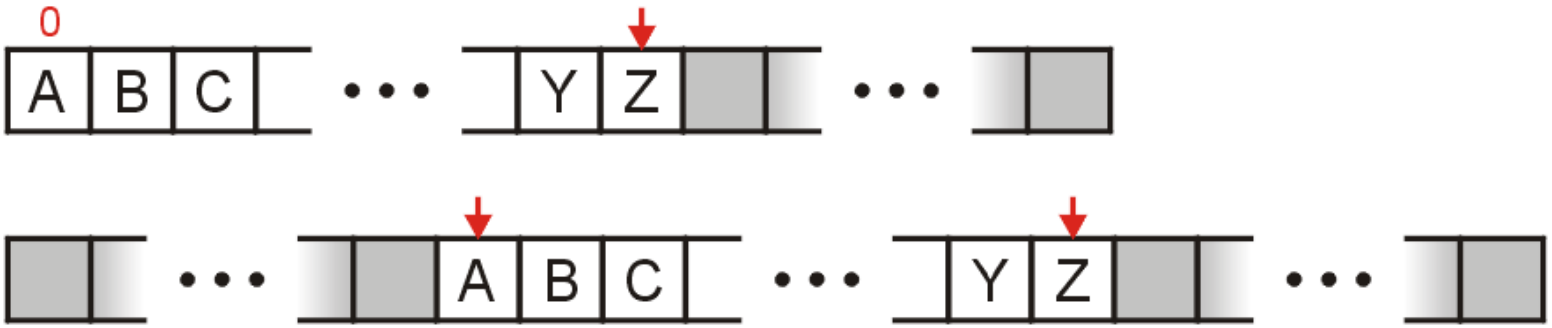
Nội dung

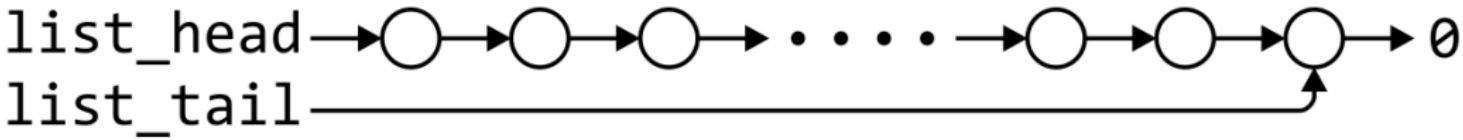
1. Nhắc lại các cấu trúc dữ liệu cơ bản
2. Giới thiệu thư viện STL và cách sử dụng các cấu trúc cơ bản
3. Một số ví dụ ứng dụng

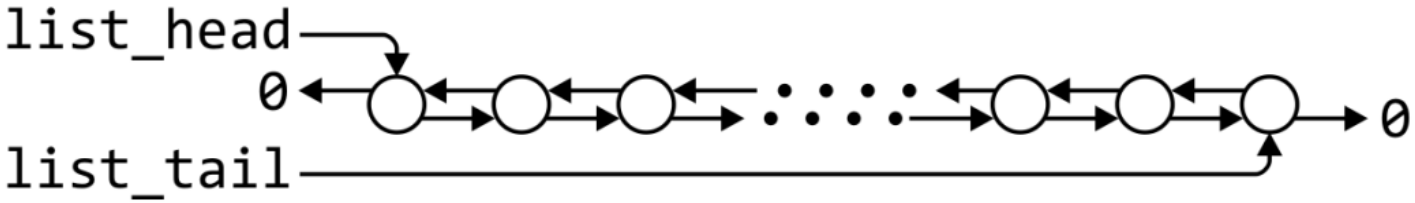
Nhắc lại các CTDL cơ bản

Các cấu trúc dữ liệu cơ bản

- Mảng và danh sách liên kết







Các cấu trúc dữ liệu cơ bản

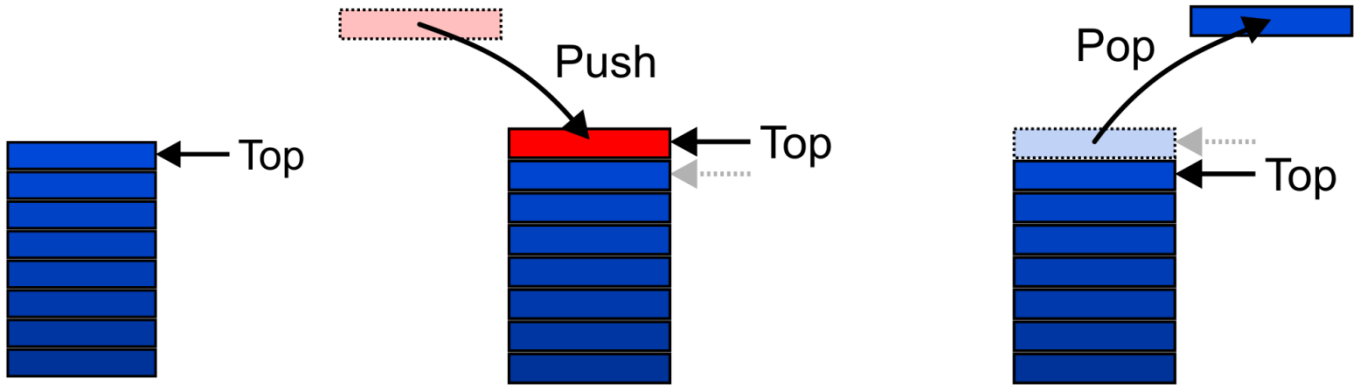
- **Mảng và danh sách liên kết**

Loại CTDL	Truy cập phần tử thứ k	Chèn/xóa phần tử		
		Đầu	Thứ k	Cuối
Danh sách liên kết đơn	$O(n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$ hoặc $\Theta(n)$
Danh sách liên kết kép				$\Theta(1)$
Mảng	$\Theta(1)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Mảng hai đầu		$\Theta(1)$		

* Giả định đã có con trỏ tham chiếu đến phần tử này

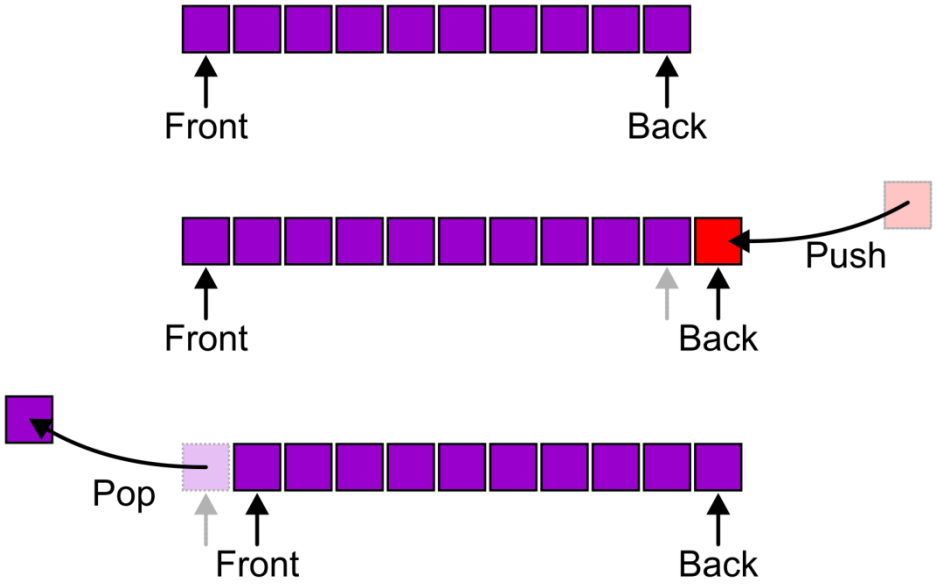
Ngăn xếp (stack)

Tuân theo thứ tự *last-in–first-out* (LIFO)

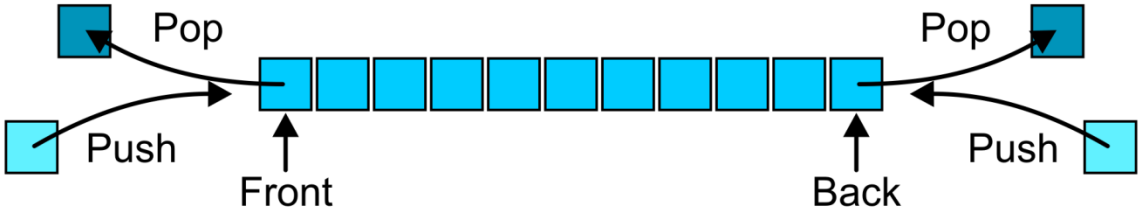


Hàng đợi (queue)

- Tuân theo thứ tự *first-in–first-out* (FIFO)



- Hàng đợi hai đầu (Deque): cho phép chèn và xóa từ hai phía



Giới thiệu thư viện STL và cách sử dụng các cấu trúc dữ liệu cơ bản

Giới thiệu thư viện STL

- STL (Standard Template Library) là thư viện chuẩn của C++, được xây dựng sẵn
- Cài đặt các cấu trúc dữ liệu và thuật toán thông dụng
- Bao gồm các lớp và hàm khuôn mẫu, cho phép làm việc với dữ liệu tổng quát
- Các thành phần chính:
  - Các lớp dữ liệu cơ bản: string, complex
  - Xuất nhập (IO)
  - Các lớp chứa (containers): pair, vector, list, deque, stack, map, set,…
  - Duyệt phần tử của các lớp chứa (iterators)
  - Một số thuật toán thông dụng: tìm kiếm, so sánh, sắp xếp,…
  - Quản lý bộ nhớ, con trỏ
  - Xử lý ngoại lệ (exception handling)

STL Containers

- **C++ 98/03 định nghĩa 3 loại container:**
  - **Sequence Containers:** `vector, deque, list`
  - **Container Adapters:** `stack, queue, priority_queue`
  - **Ordered Associative** `Containers: [multi]set, [multi]map`
- **C++11 bổ sung:**
  - `emplace{_front, _back}`
  - **Sequence Containers:** `array, forward_list`
  - **Unordered Associative Containers:** `unordered_[multi]set, unordered_[multi]map`
- **C++14 bổ sung:**
  - **Non-member** `cbegin, cend, rbegin, rend, crbegin, crend.`

STL Containers

- Màu đỏ nghĩa là có thuật toán độc lập trùng tên
- Không thể duyệt (iterate) theo queue, stack và priority_queue
- Các phương thức * không có trong multi-map

	STL containers	*Some* useful operations
	all containers	size, empty, emplace, erase
Sequence	vector<T>	[], at, clear, insert, back, {emplace,push,pop}_back
Sequence	deque<T>	[], at, emplace{,_front,_back}, insert, {,push_,pop_}back, {,push_,pop_}front
Sequence	list<T>	insert, emplace, merge, reverse,splice, {,emplace_,push_,pop_}{back,front}, sort
Sequence	array<T>	[], at, front, back, max_size
Sequence	forward_list<T>	assign, front, max_size, resize, clear, {insert,erase,emplace}_after, {push,pop,emplace}_front
Associative	set<T>, multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
Associative	map<T1,T2>, multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_set<T>, unordered_multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_map<T1,T2>, unordered_multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, hash_function
Container Adaptors	stack	top, push, pop, swap
Container Adaptors	queue	front, back, push, pop
Container Adaptors	priority_queue	top, push, pop, swap
Other	bitset (N bits)	[], count, any, all, none, set, reset, flip

Sequence Containers: vector<T>

- std::vector là mảng động (dynamic array). Hỗ trợ kiểm tra biên (bound checking) khi truy xuất phần tử `vector<T>::at()`
- Các phần tử vector được lưu liên tiếp trong bộ nhớ, do vậy các phép toán số học với con trỏ hoạt động trên vector và đảm bảo truy xuất ngẫu nhiên trong O(1).
- Thực hiện `push_back()` khi vector đã đầy sẽ dẫn tới việc cấp phát mới (reallocation).

Phương pháp truy xuất	Độ phức tạp	API hỗ trợ
Truy xuất ngẫu nhiên	O(1)	`Toán tử[]` hoặc `at()`
Thêm/xóa phần tử cuối	O(1) / O(1)	`push_back` / `pop_back`
Thêm/xóa phần tử đầu	O(N)	Không hỗ trợ API riêng
Chèn/xóa ngẫu nhiên	O(N)	`insert / erase`

Sequence Containers: vector<T>

- **Khai báo vector**

```
std::vector< <data_type> > <vector_name>;
```

- **Phương thức khởi tạo**

```
vector();
vector( size_type );
vector( size_type, const_reference );
vector( vector const & );
vector( vector && );

template < class InputIterator >
vector( InputIterator first, InputIterator last );

vector( initializer_list<value_type> );
```

Sequence Containers: vector<T>

- **Một số ví dụ khởi tạo vector**

```
// Khởi tạo vector rỗng
vector<int> vect;

// Khởi tạo vector kích thước n với các giá trị đều bằng 10
vector<int> vect(n, 10);

vector<int> vect{ 10, 20, 30 };

int arr[] = { 10, 20, 30 };
int n = sizeof(arr) / sizeof(arr[0]);
vector<int> vect(arr, arr + n);

vector<int> vect1{ 10, 20, 30 };
vector<int> vect2(vect1.begin(), vect1.end());

vector<int> vect1(10);
int value = 5;
fill(vect1.begin(), vect1.end(), value);
```

Sequence Containers: vector<T>

- Phương thức gán

```
vector &operator= ( vector const & );
vector &operator= ( vector && );
vector &operator= ( initializer_list<value_type> );
```

- **Các trình biên dịch mới cho phép thực hiện phép gán như sau:**

```
std::vector<int> v(10);
v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- **Các iterators**

```
begin  end  rbegin  rend  cbegin  cend  crbegin  crend
```

Sequence Containers: vector<T>

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> g1;
    for (int i = 1; i <= 5; i++)
        g1.push_back(i);
    cout << "Output of begin and end: ";
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";
    cout << "\nOutput of cbegin and cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i)
        cout << *i << " ";
    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << *ir << " ";
    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
        cout << *ir << " ";
    return 0;
}
```

Sequence Containers: vector<T>

- **Một số cách duyệt phần tử vector khác**

```
vector<int> vect;
for(int i = 1; i <= 5; i++)
    vect.push_back(i);

for(int i = 0; i < vect.size(); i++)
    cout << vect[i] << ' ';

for(auto &u : vect)
    cout << u << ' ';
```

Sequence Containers: vector<T>

- Kiểm tra kích thước

```
size_type size() const noexcept;
size_type capacity() const noexcept;
size_type maxsize() const noexcept;
void resize( size_type );
void resize( size_type, const_reference );
bool empty() const noexcept;
bool empty() const noexcept;
void reserve( size_type );
void shrink_to_fit();
```

Sequence Containers: vector<T>

- Truy xuất phần tử

```
reference operator[]( size_type );
const_reference operator[]( size_type ) const;

reference at( size_type );
const_reference at ( size_type ) const;

reference front();
const_reference front() const;
reference back();
const_reference back() const;

pointer data() noexcept;
const_pointer data() const noexcept;
```

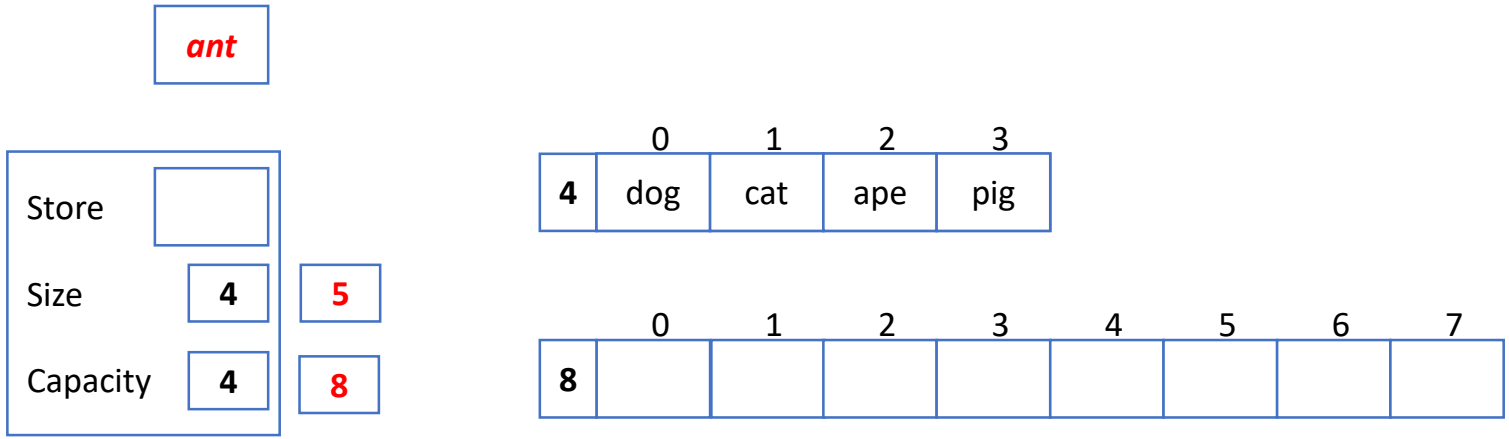
Sequence Containers: vector<T>

- Thay đổi nội dung

```
template < class Iterator >
void assign( Iterator, Iterator );

void assign( size_type, const_reference );
void assign( initializer_list<value_type> );

void push_back( const_reference );
void push_back( value_type&& );
void pop_back();

iterator insert( const_iterator position, const_reference );
iterator insert( const_iterator position, size_type n,
  const_reference );

template < class Iterator >
iterator insert( const_iterator position, Iterator first, Iterator
  last );

iterator insert( const_iterator position, value_type&& );
iterator insert( const_iterator position,
  initializer_list<value_type> );
```

Sequence Containers: vector<T>

- Cơ chế nhân đôi vùng nhớ (Doubling Policy Memory Allocation)
- Giả sử vecto đang có capacity là 4 và đã sử dụng hết
- Giờ cần push_back thêm một phần tử mới
- Khi đó một vùng nhớ gấp đôi sẽ được cấp phát, các giá trị cũ được copy sang vùng nhớ mới



Sequence Containers: vector<T>

```
int main () {
    std::vector<int>::size_type sz;
    std::vector<int> foo;
    sz = foo.capacity();
    std::cout << "making foo grow:\n";
    for (int i = 0; i < 100; ++i) {
        foo.push_back(i);
        if (sz!=foo.capacity()) {
            sz = foo.capacity();
            std::cout << "capacity changed: " << sz << '\n';
        }
    }
    std::vector<int> bar;
    sz = bar.capacity();
    bar.reserve(100);     // this is the only difference with foo above
    std::cout << "making bar grow:\n";
    for (int i=0; i<100; ++i) {
        bar.push_back(i);
        if (sz!=bar.capacity()) {
            sz = bar.capacity();
            std::cout << "capacity changed: " << sz << '\n';
        }
    }
    return 0;
}
```

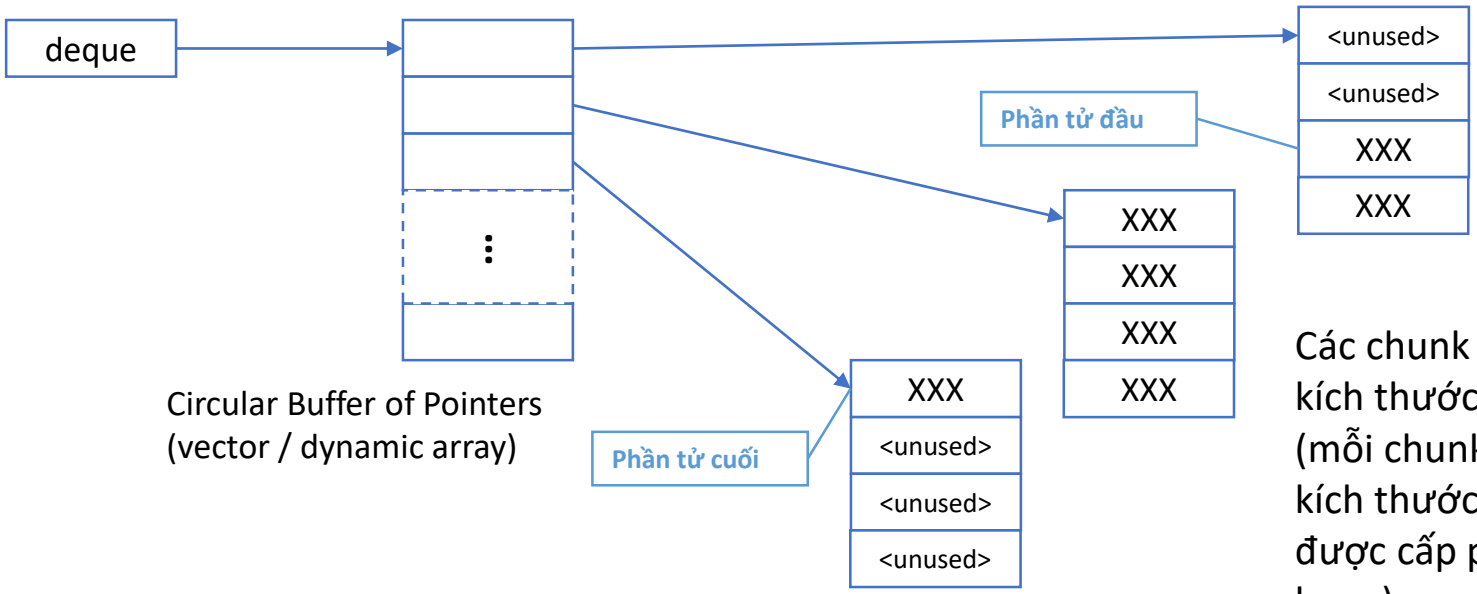
Sequence Containers: deque<T>

- Hàng đợi hai đầu "Double-Ended Queue", tương tự như vector nhưng cho phép chèn/xóa nhanh phần tử đầu.
- Truy xuất ngẫu nhiên nhanh, nhưng không đảm bảo các phần tử được lưu giữ kế tiếp nhau trong bộ nhớ
  - Do đó phép toán số học với con trỏ không hoạt động
  - Toán tử [] và at() được đa năng hóa (overloaded) để làm việc đúng

Phương pháp truy xuất	Độ phức tạp	API hỗ trợ
Truy xuất ngẫu nhiên	O(1)	operator[] or at()
Thêm/xóa phần tử cuối	O(1) / O(1)	push_back / pop_back
Thêm/xóa phần tử đầu	O(1) / O(1)	push_front / pop_front
Chèn/xóa ngẫu nhiên	O(N)	insert / erase

Sequence Containers: deque<T>

- **Cài đặt deque – Indexed and Segmented Circular Buffer**



Các chunk phần tử có kích thước cố định (mỗi chunk là mảng kích thước cố định được cấp phát trên heap)

Sequence Containers: vector vs. deque

- **Trong thực tế nên dùng `vector` hay `deque`?**
  - **Nếu chỉ cần chèn ở đầu theo kiểu FILO, hãy dùng `vector`.**
  - **Nếu cần chèn cả hai phía, hãy dùng `deque`.**
  - **Nếu cần chèn ở giữa, hãy dùng `list`.**
- **Truy xuất ngẫu nhiên tới các phần tử đều có độ phức tạp O(1), tuy nhiên truy xuất phần tử `vector` có thể nhanh hơn trong thực tế (do `deque` cần tham chiếu ngược nhiều tầng (multi-level dereferencing) mới truy xuất được)**
- **Cấp phát lại (Reallocations)**
  - **Diễn ra lâu hơn đối với `vector`.**
  - **`vector` làm mất hiệu lực các tham chiếu từ ngoài tới các phần tử, nhưng `deque` thì không.**
  - **`vector` sao chép các phần tử (mỗi phần tử có thể là các đối tượng có kích thước lớn), trong khi `deque` chỉ sao chép các con trỏ.**

Sequence Containers

```
// Vector Implementation
#include <vector>
#include <deque>
using namespace std;

int main(int argc, char* argv[]) {
    cout << "With a vector:" << endl;
    vector<int> v;
    v.push_back(4); v.push_back(3);
    v.push_back(37); v.push_back(15);
    int* p = &v.back();
    cout << *p << " " << v.at(3) << " "    //must be same
         << p << " " << &v.at(3) << endl; //must be same
    v.push_back(99);
    cout << *p << " " << v.at(3) << " "    //may be different*
         << p << " " << &v.at(3) << endl; //probably different

// Deque Implementation

    cout << "With a deque:" << endl;
    deque<int> d;
    d.push_back(4); d.push_back(3);
    d.push_back(37); d.push_back(15);
    int* p = &d.back();
    cout << *p << " " << d.at(3) << " "    //must be same
         << p << " " << &d.at(3) << endl; //must be same
    d.resize(32767);              //probably causes realloc
    cout << *p << " " << d.at(3) << " "    //must be same
         << p << " " << &d.at(3) << endl; //must be same
}
```

```
// Output below, YMMV but comments above will hold
With a vector:
15 15  0x7ff87bc039cc 0x7ff87bc039cc
15 15  0x7ff87bc039cc 0x7ff87bc039ec
With a deque:
15 15  0x7ff87c00220c 0x7ff87c00220c
15 15  0x7ff87c00220c 0x7ff87c00220c
```

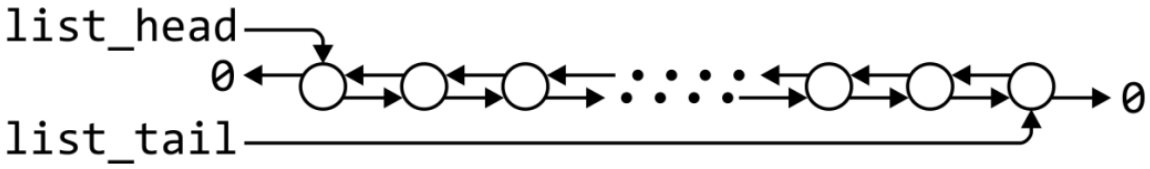
Vì p không trỏ tới v[3] nữa, nhưng giá trị cũ 15 có thể vẫn còn ở đó.

Sequence Containers: std::array (C++11)

- **Là một lớp bao (wrapper) của mảng C++, có thể xem như một `vector` kích thước cố định**
- **`std::array` vs. mảng C++**
  - **std::array không tự động chuyển thành pointer bởi trình dịch như mảng C++ thường**
  - **Hỗ trợ các hàm hữu ích**
    - **Phương thức `at()` cho phép truy xuất phần tử an toàn, kiểm tra biên**
    - **Phương thức `size()` trả về kích thước mảng được chỉ định khi khởi tạo.**
- **`std::array` vs. `std::vector`**
  - **Nếu bạn biết trước kích thước cần dùng thì hãy cố định nó!**
  - **`std::array` thường được lưu trong bộ nhớ stack hơn là bộ nhớ heap**
  - **`std::array` nhanh hơn và hiệu quả về mặt bộ nhớ hơn `std::vector`**

Sequence Containers: list<T>

- Cài đặt như danh sách liên kết kép doubly-linked list, cho phép chèn và xóa nhanh phần tử.

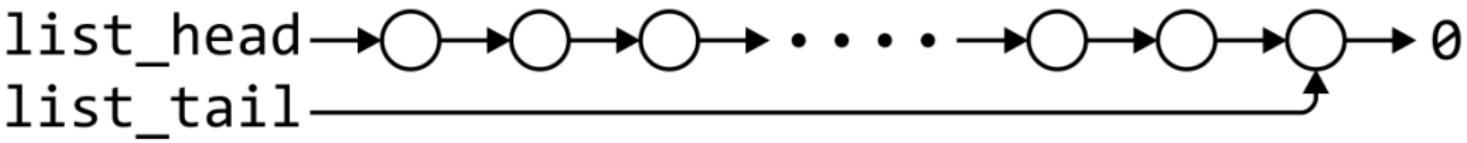


- Chỉ hỗ trợ truy xuất tuần tự thông qua iterators.
  - Không truy xuất ngẫu nhiên qua chỉ số bằng toán tử `[]` hoặc `at()`.

Phương pháp truy xuất	Độ phức tạp	API hỗ trợ
Truy xuất ngẫu nhiên	O(N)	Không hỗ trợ API riêng
Thêm/xóa phần tử cuối	O(1)	`push_back` / `pop_back`
Thêm/xóa phần tử đầu	O(1)	`push_front` / `pop_front`
Chèn/xóa ngẫu nhiên	O(1) (khi đã tới phần tử) O(N) (để tới nơi cần xóa)	`insert` / `erase`

Sequence Containers: std::forward_list (C++11)

- **Về cơ bản là danh sách liên kết đơn.**
- `std::forward_list` **vs.** `std::list`
  - **Hiệu quả hơn về bộ nhớ, các thao tác chèn/xóa nhanh hơn một chút**
  - **Không truy xuất ngay lập tức tới cuối danh sách được**
    - **Nghĩa là không có** `push_back(),back()`
  - **Không thể duyệt ngược danh sách**
  - **Không có phương thức** `size()`



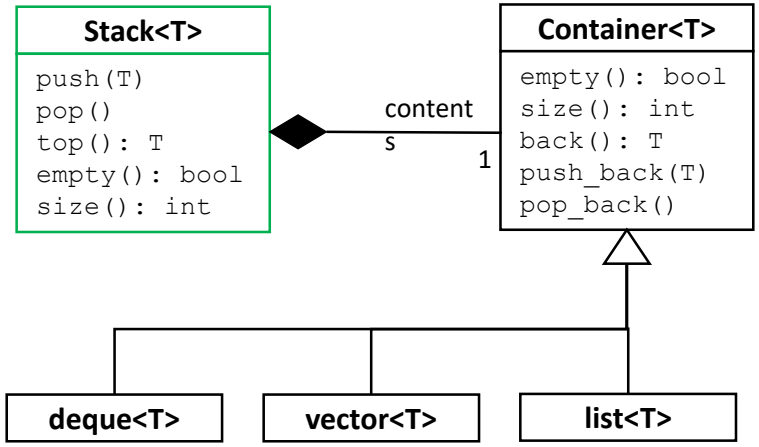
STL Container Adapters

- Thường là một lớp bao (wrapper) của sequence container để cung cấp các interface đặc thù cho các thao tác thêm/xóa phần tử.
- Có thể chỉ định trong hàm khởi tạo loại container sẽ được sử dụng để cài đặt cho adapters:
  - **`stack`**: `vector, deque(mặc định), list`
  - **`queue`**: `deque(mặc định), list`
  - **`priority_queue`**: `vector(mặc định), deque`
- Cài đặt sử dụng DP: Adapter Pattern
  - Xác định interface cần dùng (ví dụ với `stack`, ta cần `push()` và `pop()`)
  - Khởi tạo (không kế thừa!) một sequence container như là thành viên dữ liệu private của lớp để thực hiện các thao tác thực sự phía dưới (ví dụ, `vector`).
  - Định nghĩa các thao tác bằng cách ủy quyền cho các thao tác của lớp sequence container thực hiện.

STL Container Adapters – DP: Adaptor

- **Phiên bản đơn giản cài đặt STL Stack**

```
template <class T, Container = deque<T> >
class stack{
    public:
     bool empty() const;
     int size() const;
     T& top const;
     void push (const T& val);
     void pop();
    private:
     // this container is the adapter
     Container contents_;
};

void stack::push(const T& val) {
    contents_.push_back(val);
}

void stack::pop() {
    contents_.pop_back();
}
```



STL Container Adapters: stack

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

- **Các thao tác với stack**

`empty()` : **kiểm tra stack rỗng hay không**

`size()` : **Trả về kích thước của stack**

`top()` : **Trả về tham chiếu tới phần tử trên cùng của stack**

`push(g)`: **Thêm phần tử 'g' lên đỉnh của stack**

`emplace()`: **Khởi tạo và chèn phần tử lên đỉnh của stack. Hiệu quả hơn hàm push() do không mất thời gian copy đối tượng**

`pop()` – **Xóa phần tử trên cùng của đỉnh stack**

STL Container Adapters: queue

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

- **Các thao tác với queue**

`empty()`: **trả về queue rỗng hay không.**

`size()`: **trả về kích thước của queue.**

`front()`: **trả về tham chiếu tới phần tử đầu tiên trong queue**

`back()`: **trả về tham chiếu tới phần tử cuối cùng trong queue.**

`push(g)`: **thêm phần tử 'g' vào cuối queue**

`emplace()`: **khởi tạo và chèn phần tử mới vào cuối queue.**

`pop()`: **xóa phần tử đầu tiên trong queue**

STL Container Adapters: priority_queue

```
template <class T, class Container = vector<T>,
  class Compare = less<typename Container::value_type> >
class priority_queue;
```

- **Cấu trúc hàng đợi ưu tiên đảm bảo phần tử được ưu tiên nhất luôn ở đầu hàng đợi**
- **Các thao tác với priority_queue (tương tự queue)**

`empty()`: **trả về queue rỗng hay không.**

`size()`: **trả về kích thước của queue.**

`front()`: **trả về tham chiếu tới phần tử đầu tiên trong queue**

`back()`: **trả về tham chiếu tới phần tử cuối cùng trong queue.**

`push(g)`: **thêm phần tử 'g' vào cuối queue**

`emplace()`: **khởi tạo và chèn phần tử mới vào cuối queue.**

`pop()`: **xóa phần tử đầu tiên trong queue**

STL Associative Containers

- **Container liên kết có thứ tự**

  `[multi]map, [multi]set`

  - **Thứ tự các phần tử dựa trên giá trị *key***
    - **Không phải theo thứ tự chèn vào**
  - **Cài đặt sử dụng kiến trúc tương tự binary search tree => Thời gian tìm kiếm là O(log N)**
  - **Có thể duyệt các phần tử theo "thứ tự"**

- **Container liên kết không có thứ tự**

  `unordered_[multi]map, unordered_[multi]set`

  - **Không có thứ tự giữa các phần tử**
  - **Cài đặt sử dụng hash tables => Thời gian tìm kiếm là O(1)**
  - **Có thể duyệt các phần tử nhưng không theo một thứ tự cụ thể nào.**

STL Associative Containers – set<T>

- set là tập hợp các giá trị duy nhất
  - Khai báo: `set<T> s;`
  - Kiểu T phải hỗ trợ hàm so sánh thỏa mãn tính chất strict weak ordering
    - Nghĩa là chống phản xạ (anti-reflexive), phản đối xứng (anti-symmetric), bắc cầu (transitive)
    - Mặc định là toán tử `operator<`
    - Có thể dùng cho lớp do người dùng tự định nghĩa, nhưng phải định nghĩa toán tử `operator<` phù hợp cho lớp, hoặc cung cấp *functor* khi khởi tạo set.
- Set không cho phép các phần tử trùng nhau
  - Nếu cố `insert` một phần tử đã có trong `set`, thì `set` không thay đổi.
  - Kết quả trả về là một cặp `<iterator, bool>`
    - Phần tử thứ hai thể hiện phép chèn thành công hay không
    - Phần tử đầu tiên là vị trí của phần tử mới được chèn (hoặc phần tử đã tồn tại sẵn)

STL Associative Containers – set<T>

```
// Example with user-defined class and operator<
#include <algorithm>
#include <set>
#include <iostream>
#include <string>
using namespace std;

class Student {
   public:
      Student(string name, int sNum, double
gpa);
      string getName() const;
      int getSNum() const;
      double getGPA() const;
   private:
      string name_;
      int sNum_;
      double gpa_;
};

Student::Student(string name, int sNum, double
gpa) :
      name_(name), sNum_(sNum), gpa_(gpa) {}
string Student::getName() const { return name_;
}
int Student::getSNum() const { return sNum_; }
double Student::getGPA() const { return gpa_; }

bool operator== (const Student& s1, const
Student& s2) {
   return (s1.getSNum() == s2.getSNum()) &&
          (s1.getName() == s2.getName()) &&
          (s1.getGPA() == s2.getGPA())
}
```

```
bool operator< (const Student& s1, const Student& s2) {
   return s1.getSNum() < s2.getSNum();
}

ostream& operator<< (ostream& os, const Student& s) {
   os << s.getName() << " " << s.getSNum()
      << " " << s.getGPA();
   return os;
}

int main{
   // Peter and Mary have the same SNum
   Student* pJohn = new Student("John Smith", 666, 3.7);
   Student* pMary = new Student("Mary Jones", 345, 3.4);
   Student* pPeter = new Student("Peter Piper", 345, 3.1);

   set<Student> s;
   s.insert(*pJohn);
   s.insert(*pMary);
   s.insert(*pPeter);

   // Will print in numeric order of sNum
   for (auto iter = s.begin(); iter != s.end(); iter++){
      cout << iter << endl;
   }
   if ( s.find(*pPeter) != s.end() )
      cout << "Found it with set's find()!" << endl;
   if ( find( s.begin(), s.end(), *pPeter ) != s.end() )
      cout << "Found it with STL algorithm find()" << endl;
}
```

STL Associative Containers – set<T>

- **Tương đương (Equivalence) vs. đồng nhất (Equality)**
  - **Tương đương**
    - **Các phương thức tìm kiếm của container (ví dụ `find`, `count`, `lower_bound`, …) sẽ sử dụng biểu thức sau để tìm kiếm các phần tử trong container liên kết có thứ tự thậm chí ngay cả khi bạn đã định nghĩa toán tử `operator==`**

```
if( !( a < b ) && !( b < a ) )
```

  - **Đồng nhất**
    - **Các thuật toán của STL `find`, `count`, `remove_if` so sánh các phần tử bằng phép `operator==`**

STL Associative Containers – map<T>

- Tập hợp các cặp giá trị phân biệt
  - Khai báo:

```
map<T1, T2> m;
```

  - T1 là kiểu khóa (*key field type*); yêu cầu hỗ trợ hàm so sánh thỏa mãn *strict weak ordering*
    - Nghĩa là chống phản xạ (anti-reflexive), phản đối xứng (anti-symmetric), bắc cầu (transitive)
    - Mặc định là toán tử `operator<`
    - Có thể dùng cho lớp do người dùng tự định nghĩa, nhưng phải định nghĩa toán tử `operator<` phù hợp cho lớp, hoặc cung cấp *functor* khi khởi tạo set.
  - `T2` là kiểu giá trị (*value field type*); có thể là kiểu bất kỳ sao cho có thể copy hoặc gán được.

STL Associative Containers – map<T>

- **Truy vấn map tìm kiếm phần tử**
  - **Tìm kiếm theo index sẽ *chèn key mới nếu nó chưa tồn tại trong map*:**

```
if( works[ "bach" ] == 0 )
    // bach not present
```

  - **Phương pháp khác là dùng phương thức find() trả về con trỏ iterator tới cặp key/value cần truy vấn**

```
map<string, int>::iterator it;

it = words.find( "bach" );
if( it == words.end( ) )
    // bach not present
```

`end()` là giá trị iterator trỏ tới vị trí sau phần tử cuối cùng

STL Associative Containers – map<T>

- **Ví dụ Dictionary**

```
#include <iostream>
#include <map>
#include <cassert>
#include <string>
using namespace std;

// Example adapted from Josuttis
int main() {
   map<string, string> dict;

   dict["car"] = "vioture";
   dict["hello"] = "bonjour";
   dict["apple"] = "pomme";

   cout << "Printing simple
dictionary" << endl;

   for( auto it : dict ){
      cout << it.first << ":\t" <<
it.second << endl;
   }
}
```

```
// Example adapted from Josuttis
   multimap<string, string> mdict;

   mdict.insert(make_pair("car",
"voiture"));
   mdict.insert(make_pair("car", "auto"));
   mdict.insert(make_pair("car", "wagon"));
   mdict.insert(make_pair("hello",
"bonjour"));
   mdict.insert(make_pair("apple",
"pomme"));

   cout << "\nPrinting all defs of "\car"\"
<< endl;

   for(multimap<string,
string>::const_iterator
         it = mdict.lower_bound("car");
         it != mdict.upper_bound("car");
it++){

      cout << (*it).first << ": " <<
           << (*it).second << endl;
   }
```

STL Associative Containers

- `[multi]set` và `[multi]map` thường được cài đặt bằng cây đỏ đen (red-black tree)
  - Là một cây nhị phân luôn giữ cho bản thân cân bằng hợp lý bằng cách thực hiện một số thao tác khi chén/xóa phần tử.
  - Cây đỏ đen đảm bảo các thao tác lookup / insert / delete có độ phức tạp O(log N).
  - Có các phương thức tìm kiếm được tối ưu sẵn (ví dụ, `find, count, lower_bound, upper_bound`)
- Vì container được sắp xếp tự động nên không thể thay đổi trực tiếp giá trị các phần tử (vì nếu làm thế sẽ có thể gây ảnh hưởng tới thứ tự các phần tử)
  - Không có phương thức truy cập trực tiếp tới các phần tử
  - Để thay đổi một phần tử, bạn phải xóa phần tử cũ và chèn giá trị mới vào

STL Unsorted Associative Containers (C++11)

- `unordered_[multi]set` **và** `unordered_[multi]map`
- **Tương tự như các cấu trúc container liên kết có thứ tự, ngoại trừ:**
  - **Các phần tử không được sắp xếp.**
  - **Được cài đặt bằng bảng băm (hash tables), độ phức tạp O(1) với các thao tác insert / lookup / remove.**
  - **Có iterators để duyệt các phần tử trong container nhưng không theo thứ tự *đặc biệt* nào cả**

pair

```
template <class T1, class T2> struct pair;
```

- **pair chứa hai phần tử có thể thuộc hai kiểu khác nhau**
- **Truy cập phần tử thứ nhất bằng first**
- **Truy cập phần tử thứ hai bằng second**

```
pair<int, char> PAIR1;
PAIR1.first = 100;
PAIR1.second = 'G';
cout << PAIR1.first << " ";
cout << PAIR1.second << endl;
```

make_pair

```
template <class T1, class T2> pair<T1,T2>
make_pair(T1 x, T2 y);
```

- **Tạo ra một đối tượng pair**

```
pair <int,int> foo;
pair <int,int> bar;

foo = make_pair (10,20);
bar = make_pair (10.5,'A'); // ok: implicit conversion from
                            // pair<double,char>

cout << "foo: " << foo.first << ", " << foo.second << '\n';
cout << "bar: " << bar.first << ", " << bar.second << '\n';
```

make_pair

- **Khởi tạo biến pair**

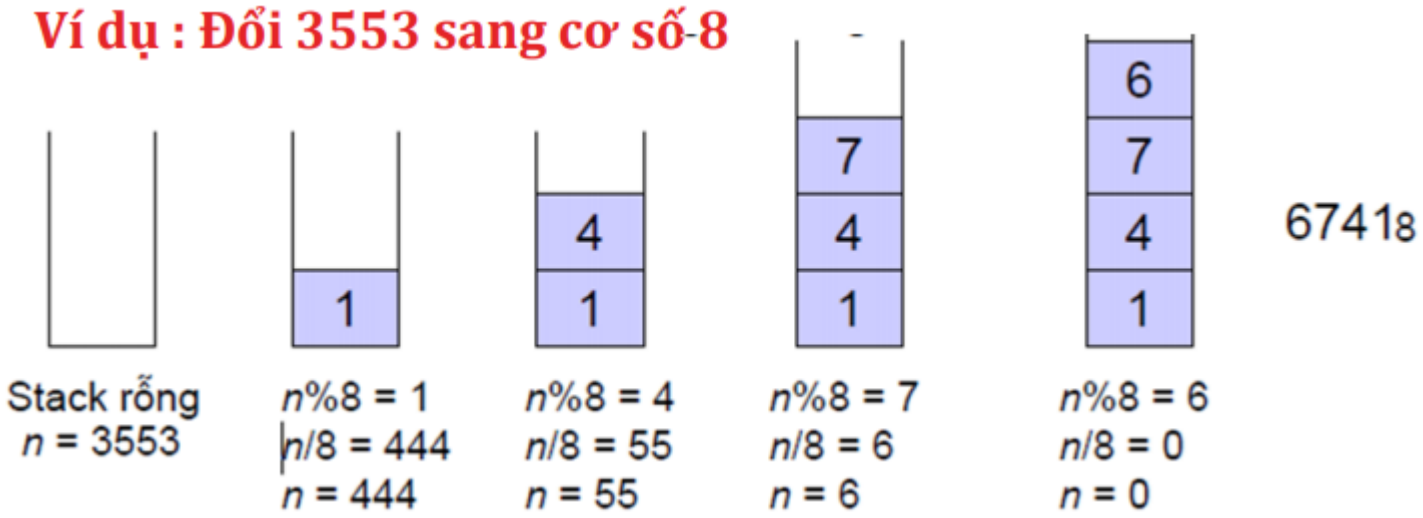
```
pair <int,int> foo = {10, 20};
pair <int,int> bar = {10.5, 'A'};

cout << "foo: " << foo.first << ", " << foo.second << '\n';
cout << "bar: " << bar.first << ", " << bar.second << '\n';
```

Một số ví dụ áp dụng

Đổi cơ số

- Dữ liệu vào: số thập phân n
- Kết quả: số hệ cơ số b tương đương

1. Chữ số bên phải nhất của kết quả = n % b. Đẩy vào Stack.
2. Thay n= n / b (để tìm các số tiếp theo).
3. Lặp lại bước 1-2 cho đến khi n = 0.
4. Rút lần lượt các chữ số lưu trong Stack, chuyển sang dạng ký tự tương ứng với hệ cơ số trước khi in ra kết quả



Đổi cơ số

```
void base_change(int n, int b) {
    char *digits = "0123456789ABCDEF";
    stack<int> stk;
    do {
        stk.push(n % b);
        n /= b;
    } while (n);

    while(!stk.empty()){
        int u = stk.top();
        stk.pop();
        cout << digits[u];
    }
}
```

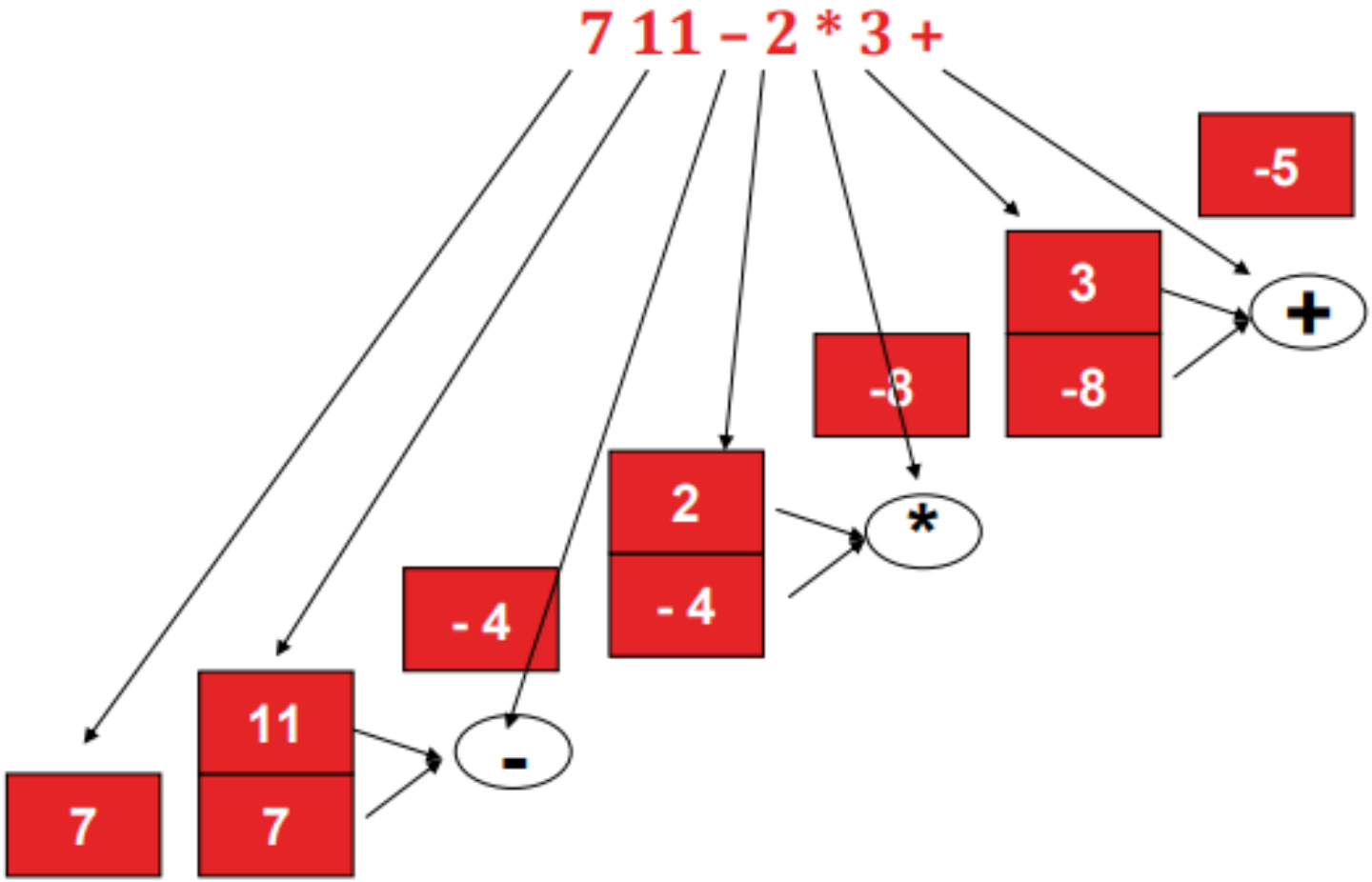
Tính toán biểu thức ký pháp hậu tố

- **Ký pháp hậu tố:**
  - **Toán hạng đặt trước toán tử**
  - **Không cần dùng các dấu ().**
- **Ví dụ:**

```
a*b*c*d*e*f => ab*c*d*e*f*
1 + (-5) / (6 * (7+8))
=> 1 5 -6 7 8 + * / +
(x/y-a*b) * ((b+x) -y )
=> x y / a b * -b x + y y ^ -*
(x*y*z -x^2 / (y*2 -z^3) + 1/z) * (x -y)
=> xy*z*x2^y2*z3^ -/ -1z/+xy -*
```

Tính toán biểu thức ký pháp hậu tố

- Ví dụ: (7 –11) * 2 + 3



Tính toán biểu thức ký pháp hậu tố

```
bool isOperator(char op) {
    return op == '+' || op == '-' ||
           op == '*' || op == '%' ||
           op == '/' || op == '^' ;
}

int compute(int left, int right, char op) {
    int value;
    switch(op){
        case '+' : value = left + right; break;
        case '-' : value = left - right; break;
        case '*' : value = left * right; break;
        case '%' : value = left % right; break;
        case '/' : value = left / right; break;
        case '^' : value = pow(left, right);
    }
    return value;
}
```

Tính toán biểu thức ký pháp hậu tố

```
int expr_eval(string expr) {
    int left, right, ans;
    char ch;
    stack<int> stk;
    for(int i = 0; i < expr.length(); i++){
        ch = expr[i];
        if (isdigit(ch)) stk.push(ch-'0');
        else if (isOperator(ch)) {
            left = stk.top();
            stk.pop();
            right = stk.top();
            stk.pop();
            ans =compute(left, right, ch);
            stk.push(ans);
        } else cout << "Expression is not correct" << endl;
    }
    return stk.top();
}
```

Water jug

- Có hai bình, một bình dung tích a lít và một bình dung tích b lít (a, b là các số nguyên dương nhỏ hơn 1000). Có một máy bơm với lượng nước không giới hạn. Cho một số nguyên dương c, làm thế nào để đong được chính xác c lít nước với số bước nhỏ nhất.
- Gọi state (x, y): lượng nước trong hai bình
- Từ trạng thái (x, y) có thể chuyển sang các trạng thái sau:
  - (x, 0)
  - (0, y)
  - (a, y)
  - (x, b)
  - (a, x + y - a) nếu x + y >= a
  - (x + y, 0) nếu x + y < a
  - (x + y - b, b) nếu x + y >= b
  - (0, x + y) nếu x + y < b
- Trạng thái cần đạt: (c, y) hoặc (x; c)

```cpp
#include <bits/stdc++.h>
using namespace std;
#define ii pair<int, int>

int cnt[1000][1000] = {};
queue<ii> q;
int a, b, c;

void push_and_count(int x, int y, int cur_steps){
    if((x < 0) || (y < 0) || (x > a) || (y > b)) return;
    if(!cnt[x][y]){
        q.push({x, y}); cnt[x][y] = ++cur_steps;
    }
}

void water_jug();

int main(){
    a = 6; b = 8; c = 4;
    water_jug();
    return 0;
}
```

```
void water_jug(){
    int x, y;
    q.push({a, b});
    cnt[a][b] = 1;
    while(!q.empty()){
        ii u = q.front();
        q.pop();
        x = u.first; y = u.second;
        if (x == c || y == c) {
            cout << cnt[x][y] - 1;
            return;
        }
        push_and_count(x, 0, cnt[x][y]);
        push_and_count(0, y, cnt[x][y]);
        push_and_count(x, b, cnt[x][y]);
        push_and_count(a, y, cnt[x][y]);
        push_and_count(a, x + y - a, cnt[x][y]);
        push_and_count(x + y, 0, cnt[x][y]);
        push_and_count(x + y - b, b, cnt[x][y]);
        push_and_count(0, x + y, cnt[x][y]);
    }
}
```

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Xin cảm ơn!