|  |
| --- |
| **BTEC FPT INTERNATIONAL COLLEGE**      **INFORMATION TECHNOLOGY**  **ASSIGNMENT 1**  **UNIT: PROGRAMMING**  STUDENT : NGUYEN DINH NHAT  CLASS : IT15101  STUDENT ID : BDAF190009  SUPERVISOR :NGUYEN HOANG ANH VU      **DaNang, June 2021** |

**ASSIGNMENT 1 FRONT SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | **Unit 19: Data Structures and Algorithms** | | |
| **Submission date** |  | **Date received (1st submission)** |  |
| **Re-submission date** |  | **Date received (2nd submission)** |  |
| **Student name** | **NGUYEN DINH NHAT** | **Student ID** | BDAF190009 |
| **Class** | **IT15101** | **Assessor name** | NGUYEN HOANG ANH VU |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.  **Student’s signature:** | | | |

**Grading grid**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **P1** | **P2** | **P3** | **M1** | **M2** | **M3** | **D1** | **D2** |
|  |  |  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **❒Summative Feedbacks: ❒Resubmission Feedbacks:** | | |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **Internal Verifier’s Comments:** | | |
| **Signature & Date:** | | |

**TABLE OF CONTENT**

[**INSTRUCTOR/ SUPERVISOR/ ASSESSOR** i](#_Toc48225899)

[**REVIEWERS** iv](#_Toc48225900)

[**ACKNOWLEDGMENTS** vii](#_Toc48225901)

[**ASSURANCE** viii](#_Toc48225902)

[**TABLE OF CONTENT** ix](#_Toc48225903)

[**LIST OF TABLES AND FIGURES** xi](#_Toc48225904)

[**LIST OF ACRONYM** xii](#_Toc48225905)

[**INTRODUCTION** 1](#_Toc48225906)

[CHAPTER 1: PLANNING A SOFTWARE DEVELOPMENT LIFECYCLE 9](#_Toc32172959)

[1.1 Software lifecycle models (P1) 9](#_Toc32172960)

[1.1.1 Waterfall model 9](#_Toc32172961)

[1.1.2 Interative model 11](#_Toc32172962)

[1.1.3 The spiral model 13](#_Toc32172963)

[1.1.4 V model 14](#_Toc32172964)

[1.2 The risk in spiral lifecycle model. (P2) 15](#_Toc32172965)

[1.3 Feasibility report (P3) 16](#_Toc32172966)

[1.4 Technical solutions (P4) 17](#_Toc32172967)

[**CONCLUSION** 7](#_Toc48225924)

[**REFERENCES** 8](#_Toc48225925)

**LIST OF TABLES AND FIGURES**

[Table 1‑1: gfdgfdgfdgfd 2](#_Toc47674358)

[Figure 1‑1: abx - ssss 2](#_Toc47674368)

[Figure 2‑1: Figure of chapter 2 4](#_Toc47674369)

[Figure 3‑1: Figure of chapter 3 5](#_Toc47674370)

**LIST OF ACRONYM**

|  |  |
| --- | --- |
| ACL | Access Control List |
| ADSL | Asymmetric Digital Subscriber Line |
| ARCNET | Attached Resource Computer Network |
| ARP | Address Resolution Protocol |
| Bit | Binary Digit |
| DHCP | Dynamic Host Configuration Protocol |
| DHCS | Dynamic Host Configuration Server |
| DNS | Domain Name System |

**INTRODUCTION**

# PLANNING A SOFTWARE DEVELOPMENT LIFECYCLE

LO1 Examine abstract data types, concrete data structures and algorithms

## Create a design specification for data structures explaining the valid operations that can be carried out on the structures (P1)

### What is data type, data structures and algorithm?

**Data** **type** là một cách phân loại dữ liệu giúp cho các trình biên dịch hiểu được các dữ liệu mà người lập trình sử dụng trong chương trình mà họ viết, với mỗi kiểu dữ liệu thì nó sẽ có các kích thước vùng nhớ khác nhau. Hầu hết, các ngôn ngữ lập trình đều cung cấp nhiều kiểu dữ liệu khác nhau. Thông thường, kiểu dữ liệu được chia thành hai loại bao gồm kiểu dữ liệu đơn giản và kiểu dữ liệu phức hợp (or abstract data types)

**Kiểu dữ liệu đơn giản** là các dữ liệu như integer, float, bool, char, string… Thông thường, các kiểu dữ liệu này được hỗ trợ trong hầu hết các ngôn ngữ lập trình như pascal, C, C++, C#, Java, JavaScript… Tên của mỗi kiểu dữ liệu này có thể sẽ khác nhau do người tạo ra ngôn ngữ lập trình định nghĩa.

Kích thước vùng nhớ, giới hạn giá trị của một số kiểu dữ liệu đơn giản trong C++ như:

|  |  |  |
| --- | --- | --- |
| **Basic data type** | **Size** | **Data range** |
| int | 4 byte | From -2147483648 to 2147483647 |
| float | 4 byte | From 1.17549e-38 to 3.40282e+38 |
| double | 8 byte | From 2.22507e-308 to 1.79769e+308 |
| char | 1 byte | Contain one characters, or one number from 0 to 255, or from -127 to 127 |
| Bool | 1 byte | 2 values: true, false. |
| long int | 4 byte | From -2,147,483,647 to 2,147,483,647 |

**kiểu dữ liệu phức hợp** (or abstract data types): là kiểu dữ liệu được hình thành từ những kiểu dữ liệu đã có (hình thành từ các kiểu dữ liệu đơn giản), hoặc từ các kiểu dữ liệu phức hợp đã được tạo trước đó. Mục đích tạo ra các kiểu dữ liệu này nhằm mục đích gần gủi với các đối tượng mà chúng ta cần phải xử lí. Trong kiểu dữ liệu phức hợp, nó được chia thành hai loại bao gồm

Nếu giá trị được lưu có cùng một kiểu dữ liệu, thì thông thường chúng là mảng

Nếu giá trị được lưu khác kiểu dữ liệu với nhau, thì thông thường chúng là dữ liệu có cấu trúc (struct).

Cấu trúc dữ liệu là cách tổ chức dữ liệu có thứ tự, có tính logic, có hệ thống sao cho việc lưu trữ dữ liệu và sử dụng dữ liệu bên trong cấu trúc đó một cách có hiệu quả.

Một số cấu trúc dữ liệu như array list, struct, Single Linked List, Doubly Linked List, stack, queue, tree, hash table, graph…

**Thuật toán** là phương pháp được tổ chức theo các bước có tính logic sao cho số lượng các bước là nhất định, nhằm mục đích giải quyết các yêu cầu của bài toán sao cho từ những input mà chúng ta có, sau quá trình xử lí theo thuật toán, chúng ta có được output đúng theo yêu cầu. Để tạo ra một thuật toán, chúng ta cần xác định được yêu cầu, input và output.

Ví dụ về thuật toán mở máy tính bàn

Input: một chiếc máy tính bàn

Output: máy tính đã được mở

Thuật toán mở máy tính

Bước 1: Kết nối điện cho máy tính

Bước 2: Nhấn nút nguồn và kết thúc

Quá trình thuật toán được thực hiện và cho đến khi kết thúc, ta có được output theo yêu cầu

### Các cấu trúc dữ liệu trừu tượng

#### Struct

Struct cho phép chúng ta định nghĩa ra một kiểu dữ liệu mới, từ các kiểu dữ liệu trừu tượng mà chúng ta đã tạo hoặc từ các kiểu dữ liệu cơ bản. Kích thước vùng nhớ của kiểu dữ liệu này phụ thuộc vào các thuộc tính ( là các thành viên bên trong struct mà ta khai báo ), có thể bằng hoặc nó lớn hơn tổng kích thước của các thuộc tính bên trong. kiểu dữ liệu tạo ra từ struct là kiểu dữ liệu mà do người lập trình viên tự định nghĩa, nó giúp cho chúng ta đóng gói các thuộc tính là các biến liên quan đến một đối tượng nào đó. Việc sử dụng kiểu dữ liệu tạo ra từ struct giúp cho chúng ta quản lí các biến được tốt hơn.

Trong C++, Kiểu dữ liệu này có thể cấp phát theo vùng nhớ stack hoặc vùng nhớ heap trên thanh RAM. Tuy nhiên cơ chế cấp phát vùng nhớ cho các thuộc tính bên trong đều được liền kề nhau. Mỗi lần cấp phát cho thuộc tính của kiểu dữ liệu struct này là 4 byte.

Trong C++ kiểu dữ liệu được khởi tạo bằng struct được tạo ra và khai báo để sử dụng như sau:

|  |  |
| --- | --- |
| Khởi tạo (định nghĩa) | Khai báo |
| typedef struct PHANSO {  int tu;  int mau;  } PhanSo; | PhanSo \*a = new PhanSo;  PhanSo b;  // Truy suat den tung thuoc tinh cua a or b  a->tu = 0;  a->mau = 0; |

#### Array

Array là một cấu trúc dữ liệu chứa các phần tử có cùng dữ liệu giống nhau, dữ liệu của các phần tử này có thể thuộc một trong các kiểu dữ liệu trừu tượng hoặc các kiểu dữ liệu đơn giản. Đối với cấu trúc dữ liệu này, trong C++ và 1 số ngôn ngữ khác, đều có mảng tĩnh và mảng động. Đối với mảng tĩnh cần biết số lượng trước khi thực hiện. Việc cấp phát cho vùng nhớ này ngay khi chương trình hoạt động.

Cú pháp khai báo mảng tĩnh trong C++ là:

|  |
| --- |
| <data type><name variable>[<amount>];  //Example: int myNumbers[100];  // myNumbers[0] = 1; myNumbers[1] = myNumbers[0]; |

Trong quá trình sử dụng mảng tĩnh, số lượng phần tử phải là một giá trị số nguyên dương nhất định hoặc là một hằng số có giá trị là số nguyên dương. Các vùng nhớ cấp phát cho mảng tĩnh được lưu tại vùng nhớ stack. Việc cấp phát mảng và thu hồi là do CPU thực hiện quá trình này.

Đối với mảng động, số lượng mảng có thể là một biến chứa giá trị số nguyên dương, một giá trị số nguyên dương nhất định hoặc một hằng số có giá trị là số nguyên dương. Vùng nhớ cấp phát cho mảng động được lưu tại vùng nhớ heap trên thanh ram. Việc giải phóng, cấp phát vùng nhớ này do người lập trình thực hiện. Tuy nhiên, trên thực tế khi chương trình kết thúc, vùng nhớ này sẽ bị thu hồi và giải phóng để cung cấp cho các phần mềm khác nếu phần mềm nào đó cần sử dụng.

Trong C++, Cấu trúc khai báo và sử dụng mảng động được thực hiện như sau:

|  |
| --- |
| <data type> \*<name variable> = new <data type>[<amount>];  //Example: int \*myNumbers = new int[100];  //or: int count = 10; int \*myNumbers = new int[count];  //myNumbers[0] = 1; myNumbers[1] = myNumbers[0]; |

#### Array list

Array list là một cấu trúc dữ liệu, có thể gọi là siêu mảng động, nó có các tính chất và các chức năng lưu trữ tương tự như mảng động và mãng tĩnh. Tuy nhiên, đối với Array list, việc cấp phát lần đầu, cấp phát lại bằng cách thêm phần tử hoặc giảm bớt phần tử được thực hiện và hỗ trợ, giúp cho lập trình viên sử dụng, thao tác và lưu trữ dữ liệu một cách dễ dàng hơn. Thông thường, Array list có thể được hổ trợ đối với một số ngôn ngữ lập trình như các kiểu dữ liệu cơ bản. Tuy nhiên Array list có thể là được tạo ra bởi một thư viện, hay API library trong một số ngôn ngữ khác.

Trong C++, đối tượng vector trong thưc viện vector có cấu trúc dữ liệu là array list.

Việc khởi tạo để sử dụng được thực hiện bằng cách như sau:

|  |
| --- |
| //vector< <data type> > <name vector>;  //Example: vector<int> myVector; |

Thông thường, trong các ngôn ngữ lập trình bậc cao, Array list được hổ trợ thêm các function nhằm mục đích thao tác các dữ liệu bên trong Array list như thêm, xóa, sửa...

#### Single Linked List

Danh sách liên kết đơn là một cấu trúc dữ liệu giúp giải quyết các vấn đề liên quan đến việc cấp phát một vùng nhớ lớn liền kề nhau. Đối với Array hay arraylist, khi chúng yêu cầu cấp phát vùng nhớ, các ô nhớ trong vùng nhớ này được cấp phát sẽ là một vùng nhớ liền kề, chứa một khoảng lớn. Tuy nhiên, với danh sách liên kết đơn, các node tạo ra chỉ cần một vùng nhớ nhỏ và các node này được liên kết với nhau thông qua con trỏ, chính vì thế mà các vùng nhớ cấp phát cho các node của linked list không nhất thiết phải liền kề nhau. Tuy nhiên, trong thực tế, khi cấp phát một vùng nhớ cho node, nó sẽ cấp phát tại địa chỉ lớn nhất hiện tại trên vùng nhớ stack của thanh RAM. Trong C++, danh sách liên kết đơn được tạo ra từ kiểu dữ liệu struct, mỗi node thuộc kiểu dữ liệu được định nghĩa từ struct này. Trong các node, sẽ có 2 thành phần bao gồm một pointer trỏ đến node tiếp theo, và thành phần thứ hai là data

|  |
| --- |
| // create structure data  // typedef struct PHANSO {  PHANSO \* pointer;  int data;  } PhanSo;  // definition single linked list data type  // phanSo \*a = new PhanSo;  // phanSo \*b = new PhanSo;  // a->pointer = b;  // access data of node b from node a  // a->pointer->data = 2; |

#### Doubly Linked List

Danh sách liên kết đôi được phát triển từ danh sách liên kết đơn, giúp giải quyết được vấn đề dễ dàng truy cập bắt đầu từ phần tử đầu tiên hoặc bắt đầu từ phần tử cuối cùng vào các node ở giữa danh sách một cách dễ dàng hơn.

|  |
| --- |
| // create structure data  // typedef struct PHANSO {  PHANSO \* pointerHead;  PHANSO \* pointerTail;  int data;  } PhanSo;  // definition single linked list data type  // phanSo \*a = new PhanSo;  // phanSo \*b = new PhanSo;  // a->pointerHead = b;  // access data of node b from node a  // a->pointerHead->data = 2;  // b->pointerTail = a;  // access data of node a from pointerTail of node b  // a->pointerHead->pointerTail->data = 2; |

#### Stack

Stack là một cấu trúc dữ liệu mà bản chất của nó sinh ra từ danh sách liên kết đơn. Và cơ chế của stack segment được tạo ra dựa theo cấu trúc dữ liệu stack này. Cấu trúc dữ liệu Stack được hổ trợ hầu hết trên các ngôn ngữ lập trình và cung cấp các hàm đi kèm nhằm mục đích giúp cho lập trình viên thao tác lên cấu trúc dữ liệu này dễ dàng hơn. Ngoài ra, cơ chế của Stack là LIFO. Nó nghĩa là thành phần nào được đưa vào sau thì lấy ra trước. Một ví dụ điển hình thực tế khi nói về cơ chế của stack là việc sắp các dĩa chồng nhau, nếu dĩa nào vào sau thì nó có thể lấy ra trước…

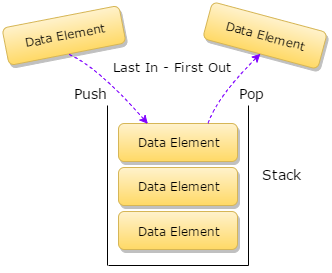


Figure : Describe Stack abstract data type

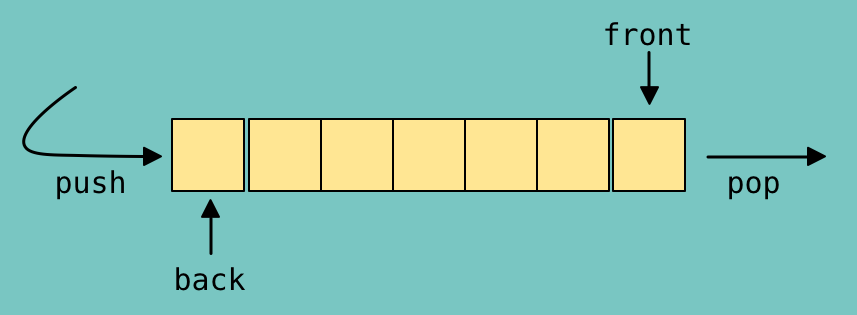
Các hàm hổ trợ xử lí, thao tác với dữ liệu trong stack được trình bày dưới đây:

Trong C++, cấu trúc dữ liệu stack được hổ trợ các hàm sau:

|  |  |  |
| --- | --- | --- |
| Function | Meaning | Example |
| push | Thêm một node vào cuối stack có giá trị là x | myStack.push(x);  // add new node, it has data is x |
| pop | Lấy ra dữ liệu của node cuối cùng và xóa node đó ra khỏi stack | myStack.pop();  // delete last node. |
| top | Lấy dữ liệu của node cuối cùng nhưng không xóa node đó ra khỏi stack | myStack.top();  // return data of last node |
| empty | Kiểm tra stack đó có rỗng hay không, trả về kiểu dữ liệu Boolean | myStack.empty();  // check myStack have empty or not |
| size | Trả về kích thước là số lượng các node trong stack | myStack.size();  // return 1 integer is amount node in myStack. |
| swap | Hoán đổi toàn bộ các node của 2 stack cho nhau | myStack.swap(yourStack); |

#### Queue

Queue là một cấu trúc dữ liệu mà nó được tạo ra bằng cách cập nhật từ danh sách liên kết đơn. Hầu hết, các ngôn ngữ lập trình đều hỗ trợ Queue và cung cấp các function giúp lập trình viên thao tác với dữ liệu bên trong Queue một cách dễ dàng hơn. Queue khác Stack ở cơ chế FIFO. Nó có nghĩa là thành phần nào vào trước thì sẽ được lấy ra trước. Một ví dụ điển hình cho Queue là những khách hàng sắp xếp thứ tự mua hàng, nếu khách hàng nào vào trước thì sẽ mua và đi ra trước. Những người nào vào sau thì sẽ ra sau…



Các hàm hổ trợ thao tác và xử lí với dữ liệu bên trong Queue được trình bày ở dưới đây:

Trong C++, cấu trúc dữ liệu stack được hổ trợ các hàm sau:

|  |  |  |
| --- | --- | --- |
| Function | Meaning | Example |
| front | Lấy ra node đầu tiên của queue | myQueue.front();  // return value of first node |
| back | Lấy ra node cuối cùng của queue | myQueue.back();  // return value of last node |
| push | Thêm một node vào cuối queue có giá trị x | myQueue.push(x);  // add node has value x |
| Empty | Kiểm tra queue có rỗng hay không | myQueue.Empty();  // check myQueue is empty or not |
| pop | Xóa node đầu tiên của queue | myQueue.pop();  // delete first node |
| swap | Hoán đổi các node của 2 queue cho nhau | myQueue.swap(); |
| size | Trả về số lượng các node trong queue | myQueue.size();  // return amount node of myQueue |

#### Tree

Tree là một cấu trúc dữ liệu mà trong cây nó có ít nhất 2 thành phần có chức năng liên kết đến các node khác. Trong tree, có một cấu trúc dữ liệu chi tiết hơn gọi là cây nhị phân. Cây nhị phân được tạo ra dựa trên cơ chế của danh sách liên kết, tuy nhiên các thành phần bên trong cây có thêm một thuộc tính link nhằm liên kết thêm một node khác. Cứ mỗi node trong cây nhị phân, nó sẽ trỏ đến 2 node khác và tại mỗi node thì chỉ quản lí một data nào đó. Hầu hết cấu trúc dữ liệu này không được tạo ra sẵn để lập trình viên sử dụng mà người lập trình viên cần tạo ra chúng. Một số hàm cần xây dựng khi thao tác với cấu trúc dữ liệu này là thêm, xóa, sửa, và xuất giá trị giữa các node. Ngoài ra, cây nhị phân cân bằng là một cấu trúc dữ liệu được cập nhật từ cây nhị phân sao cho các node thuộc một tầng không bị chênh lệnh quá 1 node con ở giữa các node. Có nghĩa là, tại một node, độ cao của cây con trái và độ cao của cây con phải của node nào đó bị chênh lệnh không quá một.

#### Hash table

Hash table là một cấu trúc dữ liệu được sinh ra để giải quyết vấn đề tìm kiếm một phần tử nào đó trong danh sách các phần tử, đáp ứng được độ phức tạp của thuật toán tìm kiếm là O(1). Thay vì phải tạo ra các hàm tìm kiếm hay các thuật toán tìm kiếm cho các hàm đó, chúng ta sẽ tổ chức dữ liệu để tạo ra một cấu trúc sao cho khi ta biết được dữ liệu cần kiểm tra xem trong danh sách đó có dữ liệu cần tìm hay không, ta chỉ cần truy vết từ dữ liệu đó, thì có thể biết được nó có hay không. Với hash table, mỗi phần tử sẽ có 2 thuộc tính bao gồm key and value. Phụ thuộc vào key chúng ta sẽ biết được value. Ý nghĩa key và value nó cũng tương tự như index và value của 1 element trong array. Tuy nhiên, key này khác với index ở chỗ nó được xác định và tạo ra thông qua value.

Ta có thể xét ví dụ đơn giản về has table như sau:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | **5** | **4** | **1** | **7** | **8** | **3** | **9** |
| **Index** | **0** | **1** | **2** | **3** | **4** | **5** | **6** |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | -1 | **2** | -1 | **5** | **1** | **0** | -1 | **3** | **4** | **6** |
| **Index** | 0 | **1** | 2 | **3** | **4** | **5** | 6 | **7** | **8** | **9** |

Với bảng 1 là các giá trị mà người dùng nhập vào và với bảng 2 là bảng hash, sử dụng các giá trị của bảng 1 làm chỉ số để đánh dấu cho bảng hash và nếu chỉ số nào trong bảng hash trùng với giá trị của bảng 1 thì giá trị của bảng 1, thì giá trị trong bảng hash sẽ lưu index tương ứng của giá trị tại bảng 1 đó.

Tuy nhiên trong một số trường hợp, các dữ liệu là các giá trị bên trong mảng có thể nó giống nhau, thì sẽ gây ra việc đụng độ(conflict).

Ta có thể xét ví dụ 2 như sau:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | **5** | **4** | **8** | **7** | **8** | **3** | **9** |
| **Index** | **0** | **1** | **2** | **3** | **4** | **5** | **6** |

Ở bảng trên đây, ta có 2 giá trị 8 với index của elements trong array này là 2 và 4. Để giải quyết được vấn đề đụng độ này, thì với 1 phần tử của mảng hash ta có thể sử dụng một trong 3 phương pháp sau:

Với mỗi phần tử của mảng hash, ta sử dụng queue hoặc stack để lưu index của mảng hash

Với mỗi phần tử trùng lặp nhưng khác index của ví dụ 2 thì ta sẽ đưa ra các công thức nhằm hạn chế việc đụng độ giữa các giá trị nhất có thể. Ví dụ như đối với số, chúng ta sẽ sử dụng các số nguyên tố để chia ( giá trị của mảng mà người dùng nhập % số nguyên tố => index của bảng hash lưu giá trị của mảng mà người dùng nhập).

Với mỗi phần tử của mảng hash, ta có thể sử dụng danh sách liên kết đơn để lưu index của mảng hash.

Việc tạo ra cấu trúc dữ liệu này, nó không có một cấu trúc nhất định mà nó phụ thuộc vào dữ liệu đầu vào để thiết kế sao cho cấu trúc dữ liệu hash table phù hợp với dữ liệu đó.

#### Graph

Cấu trúc dữ liệu đồ thị là một cấu trúc dùng để biểu diễn tập hợp các node, trong đó các cặp node này sẽ được liên kết bởi các link. Trong đó, các node này sẽ được gọi là vertices và các link kết nối giữa 2 node gọi là edges. Tóm lại, một đồ thị sẽ tập hợp (V,E) trong đó, V là tập hợp các đỉnh trong đồ thị và E là tập hợp các cạnh mà kết nối các cặp điểm lại với nhau.

Trong cấu trúc dữ liệu đồ thị, nó được chia thành 2 loại bao gồm: Directed graphs and Undirected graphs.

Directed graphs: là một đồ thị mà ta có thể xác định được đỉnh đầu và đỉnh kết thúc và xác định được hướng của các cạnh…

A graph is undirected: là một đồ thị có thể chứa một hoặc nhiều điểm cô lập, các cạnh của đồ thị không xác định được hướng, có nghĩa là chúng có thể tạo ra 2 hướng đi ngược nhau (giống như đường hai chiều).

2 Cách để biểu diễn đồ thị này bao gồm danh sách kề và ma trận kề

1. cách để liệt kê các điểm trên đồ thị hoặc điều hướng biểu đồ bao gồm:

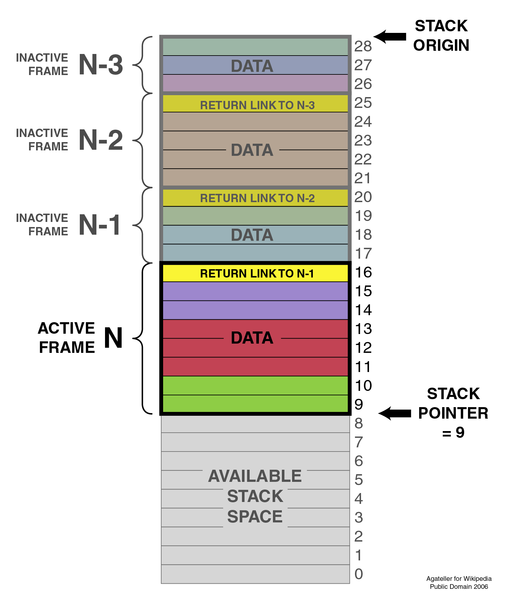
Tìm kiếm đầu tiên theo độ sâu và tìm kiếm đầu tiên theo độ rộng.

## Determine the operations of a memory stack and how it is used to implement function calls in a computer (P2)

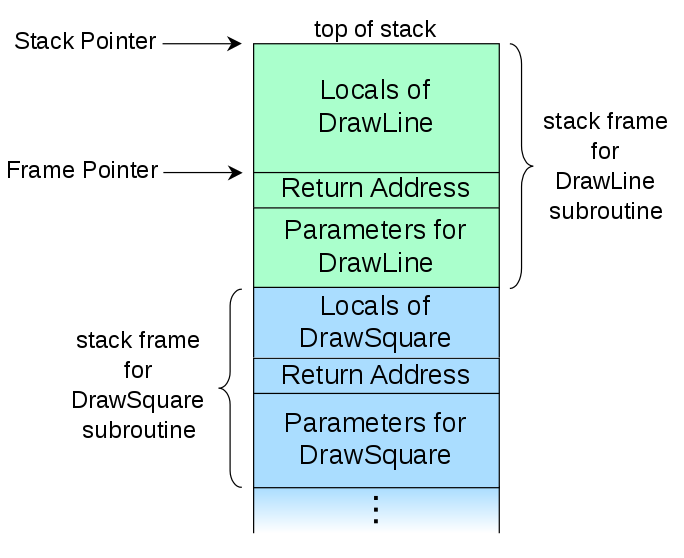


Memory stack chính là một kiến trúc, là một vùng nhớ trong máy tính. Khi một hàm trong một chương trình nào đó được thực thi, nó có nhiệm vụ lưu trữ một số dữ liệu, trạng thái của hàm được gọi. Khi hàm đó hoàn thành quá trình công việc, stack sẽ xóa những tiến trình của hàm đó ra khỏi stack. Vùng nhớ stack được hoạt động giống như cơ chế của cấu trúc dữ liệu stack. Nghĩa là, dữ liệu được lưu tại vùng nhớ này khi được thêm vào hoặc xóa đi cũng thực hiện theo cách thức LIFO.

Khi chương trình hoạt động, hàm main sẽ được gọi đầu tiên, các biến cục bộ của hàm main được lưu trữ tại vùng nhớ stack, (điều này có thể hiểu là stack push data of local variables in main function). Tại vùng nhớ stack, các biến cục bộ hoặc các tham số của hàm sẽ được cấp phát để lưu trữ dữ liệu tại vùng nhớ này.



Callstack là



Nếu trong hàm main ta gọi tới hàm khác (giả sử là hàm A) thì nó sẽ lưu hàm và tham số truyền hàm A vào stack. khi hàm A thực hiện xong nó sẽ pop hàm A ra khỏi vùng nhớ stack. Và khi hàm main thực hiện xong, nó cũng sẽ pop hàm main ra khỏi vùng nhớ stack. Để hiểu rõ hơn ta có thể xét ví dụ sau:

Ví dụ được soạn ra trong chương trình viết bằng ngôn ngữ lập trình C++:

|  |
| --- |
| #include <iostream>  using namespace std;  int Square(int x) {  return x \* x;  }  int SquareOfSum(int x, int y) {  return Square(x + y);  }  int main() {  int a = 2, b = 5;  cout << SquareOfSum(a, b) << endl;  return 0;  } |

Đầu tiên, chương trình sẽ vào hàm main để thực hiện trước, khi gặp câu lệnh khai báo biến a và b, vùng nhớ stack sẽ push hai biến a và b vào. Khi đó, vùng nhớ stack trên thanh ram sẽ chứa như sau:

|  |  |
| --- | --- |
| Acction | Memory stack |
| Push(a and b) | main(), a, b |

Tiếp theo, khi chương trình gặp câu lệnh cout để in ra màn hình kết quả của hàm SquareOfSum trả về. Để in ra được kết quả mà hàm SquareOfSum trả về, chương trình cần gọi đến hàm SquareOfSum để thực hiện. Khi đó vùng nhớ stack sẽ push hai tham số được truyền vào hàm, đồng thời nó cũng push hàm SquareOfSum vào vùng nhớ stack. Khi đó, vùng nhớ stack trên thanh ram sẽ chứa như sau:

|  |  |
| --- | --- |
| Acction | Memory stack |
| Push(SquareOfSum function, x and y) | SquareOfSum(), x and y |
| Push(main function, a and b) | main(), a, b |

Tuy nhiên, sau khi hàm SquareOfSum được lưu trữ, chương trình tiếp tục thực hiện các câu lệnh trong hàm này. Ta thấy rằng, trong hàm SquareOfSum gọi đến một hàm khác là Square. Trước khi đi đến hàm Square làm việc, vùng nhớ stack đã lưu hàm Square và tham số x trong hàm này vào. Khi đó trong vùng nhớ stack sẽ như sau:

|  |  |
| --- | --- |
| Acction | Memory stack |
| Push(Square and x) | Square and x |
| Push(SquareOfSum function, x and y) | SquareOfSum(), x and y |
| Push(main function, a and b) | main(), a, b |

Trong hàm Square, sau khi thực hiện tính toán x\*x thì nó trả về giá trị tính toán của biểu thức x\*x này thông qua tên hàm. Sau khi trả về giá trị đó, vùng nhớ stack sẽ pop hàm Square và tham số x ra khỏi vùng nhớ. Khi đó, trong vùng nhớ stack chỉ còn như sau:

|  |  |
| --- | --- |
| Acction | Memory stack |
| Push(SquareOfSum function, x and y) | SquareOfSum(), x and y |
| Push(main function, a and b) | main(), a, b |

Tiếp tục, sau khi hàm Square trả về giá trị, Hàm SquareOfSum cũng trả về giá trị từ hàm Square. Sau khi thực hiện xong hàm SquareOfSum, vùng nhớ stack sẽ pop hàm SquareOfSum và 2 tham số x, y ra khỏi vùng nhớ. Khi đó vùng nhớ stack chỉ còn như sau:

|  |  |
| --- | --- |
| Acction | Memory stack |
| Push(main function , a and b) | main(), a, b |

Sau khi thực hiện xong hàm SquareOfSum, thì chương trình đã có kết quả của hàm này trả về. Lúc đó, hàm cout in ra màn hình giá trị từ hàm SquareOfSum trả về. sau khi thực hiện xong hàm cout. Chương trình return 0 để thông báo chương trình hoạt động thành công. Trong C++, giá trị 0 này tương ứng với hằng số có tên là EXIT\_SUCCESS. Mục đích return 0 để thông báo cho hệ điều hành biết chương trình đã hoạt động thành công. Đồng thời, giải phóng đi các vùng nhớ đã cấp phát cho chương trình.

**CONCLUSION**

After completing this report, I have gained…..

**REFERENCES**

1. Nguyễn Kim Sơn, Phạm Hùng Vân, Nguyễn Bảo Sơn và cộng sự (2010). Đột biến gen mã hóa EGFR trong ung thư phổi. Tạp chí nghiên cứu y học, 3, 30-37.

2. Amanda B.R, Donna P.A, Robin J.L et al (2008). Total prostate specific antigen stability confirmed after long-term storage of serum at -80C. J.Urol, 180(2), 534-538.

Cách ghi các tài liệu tham khảo tại:

<https://journal.hiu.vn/vi/cach-trich-dan-tai-lieu/cach-trich-dan-tai-lieu-tham-khao>