Basic Data Structure

Instructor

Pathorn Tengkiattrakul, pathorn.teng@gmail.com

Today's topics

- Linked List
- Stack
- Queue
- Hash Table

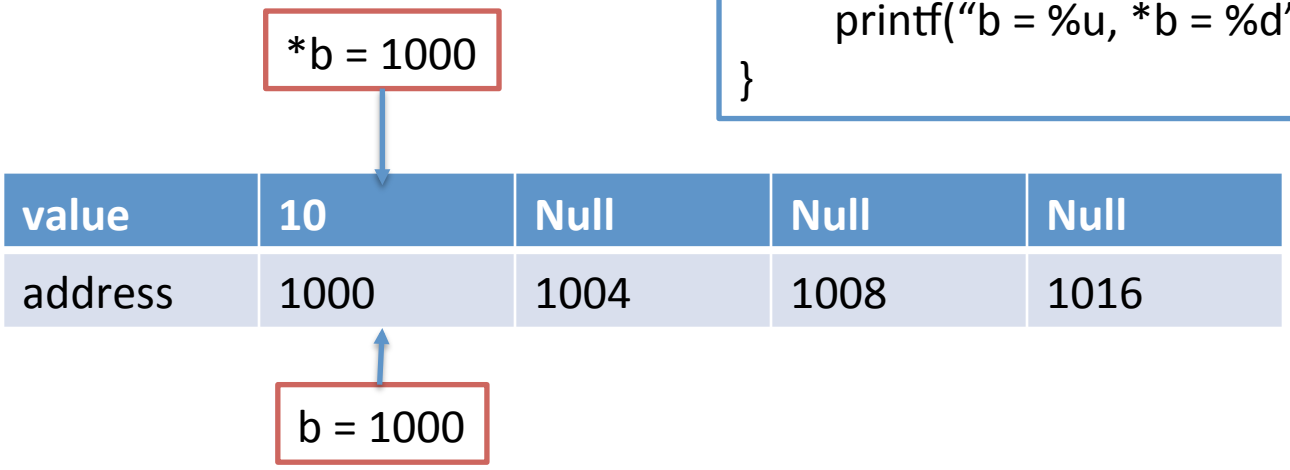
Why Data Structure?

- โปรแกรมส่วนใหญ่ทำงานกับข้อมูล
- กระบวนการสำคัญที่มีผลต่อประสิทธิภาพของโปรแกรมคือ
  - การเพิ่มข้อมูล (Insert)
  - หาค้นหาข้อมูล (Search)
    - ค้นหาเพื่อแก้ไข (Delete/Edit)
    - ค้นหาเพื่อนำมาแสดง (View)
- การเข้าใจการทำงานของ data structure แต่ละเภทช่วยให้การเขียนโปรแกรมมีประสิทธิภาพดีขึ้น

ทบทวน pointer

- ตัวแปรทุกตัวต้องการที่อยู่ใน (memory address) การเก็บข้อมูล
- Pointer คือตัวแปรที่ใช้เก็บที่อยู่ (memory address)

```
#include<stdio.h>
main(){
        int a = 10;
        int *b;
        b = &a;
        printf("b = %u, *b = %d",b,*b);
}
```



Array

- Array เป็น data structure พื้นฐานที่ C ได้เตรียมไว้
- ตัวแปร array ถือเป็น data structure ชนิดหนึ่งที่ทำงานได้เร็ว
  - N = จำนวนสมาชิก
  - Insert data ความเร็ว = n/2 หรือ 1
  - Search data ความเร็ว = n/2
  - Delete data ความเร็ว n/2 หรือ 1

Array Limitation?

- Array ทำการจอง memory ไว้ก่อนโดยไม่ได้ใช้งาน (resource wasted)
- ไม่สามารถเพิ่มขยายพื้นที่ได้ในภายหลัง (size fixed)
- Insert/Delete ใน array ไม่มีประสิทธิภาพ (ต้องย้ายสมาชิก)
- สรุป ข้อเสียของ array สองอย่าง
  - Fixed size
  - Inefficient insert and delete

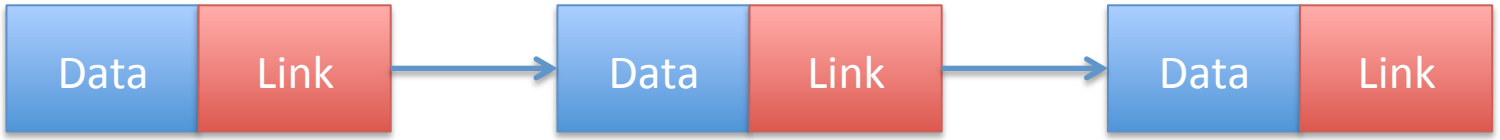
Linked list

- Linked List เป็น data structure พื้นฐานที่เป็นที่รู้จักและใช้งานได้หลากหลาย
- ภาษา C ไม่มีตัวแปรประเภท Linked list (Java มี) ดังนั้น นักพัฒนาโปรแกรมจึงต้องเขียน Linked list ขึ้นมาเอง (หรือหา library มาใช้งาน)
- Linked list มีสองประเภทหลักคือ
  - Single linked list
  - Double linked list

หลักการทำงานของ Linked List

- Linked List ประกอบไปด้วย 2 อย่างคือ
  - Node ทำหน้าที่ในการเก็บข้อมูล
  - Link ทำหน้าที่ชี้ไปยัง node ถัดไป
- ในภาษา C เรามักใช้ struct ในการสร้าง Node

```
typedef struct node {
    int data;
    struct node * next;
} node_t;
```

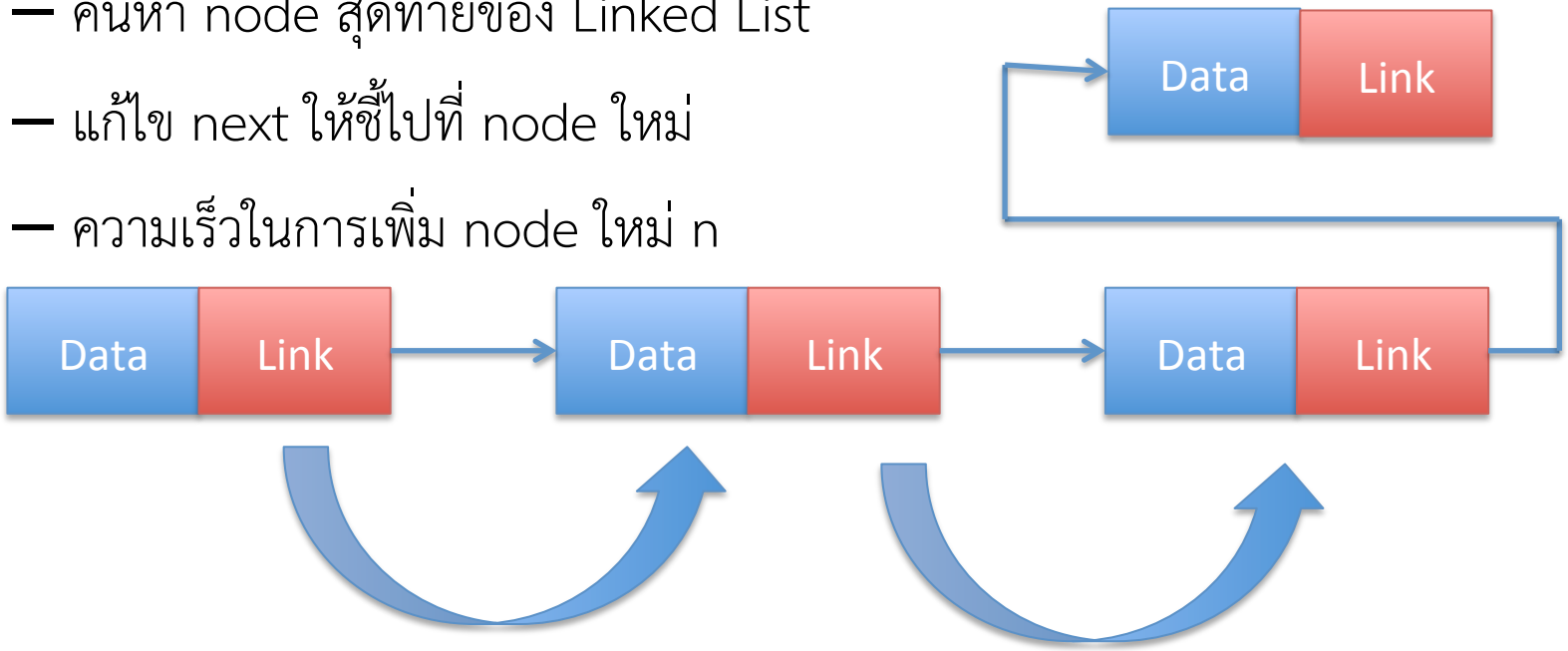


ฟังก์ชันที่จำเป็นสำหรับ data structure

- Function สำคัญสำหรับ Linked List ประกอบไปด้วย
  - Insert ใช้ในการเพิ่ม node ใหม่ใน data structure
  - Find/Search/Get ใช้ในการค้นหาข้อมูล (ค้นหา node)
- การเลือกใช้ data structure ไหนนั้นขึ้นอยู่กับชนิดของโปรแกรม
  - เรียกดูข้อมูลบ่อย เลือก data structure ที่ find/search/get ทำงานเร็ว
  - เพิ่มข้อมูลบ่อย เลือก data structure ที่ insert ทำงานเร็ว

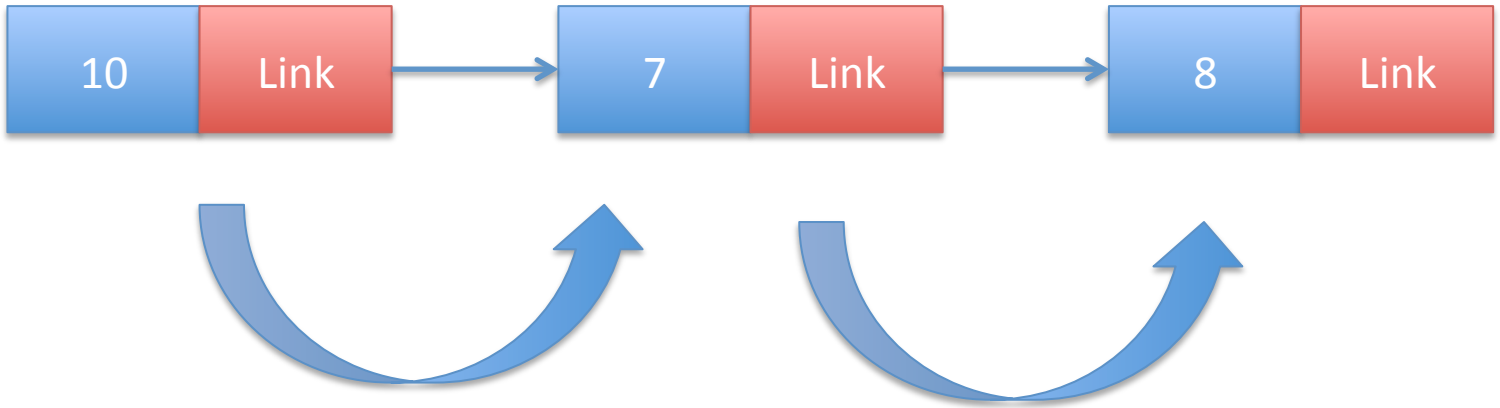
Insert Linked List

- คำสั่ง insert ใน Linked List นั้นเป็นการเพิ่มข้อมูลลงใน data structure
- การทำงานของ Insert
  - ค้นหา node สุดท้ายของ Linked List
  - แก้ไข next ให้ชี้ไปที่ node ใหม่
  - ความเร็วในการเพิ่ม node ใหม่ n



Find/Search/Get

- สำหรับฟังก์ชัน find/search/get นั้น Linked List จะทำการค้นหาทีละ node จนกว่าจะเจอ node ที่ต้องการ
- ความเร็วในการค้นหา n/2



ทบทวนการ reference point

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node {
	int data;
	struct node *next;
} node_t;

int main(){
	node_t *pointer;
	node_t node;
	node.data = 20;
	pointer = (node_t*) malloc(sizeof(node_t));
	pointer->data = 10;
	printf("pointer->data = %d\n", pointer->data);
	free(pointer);
	pointer = &node;
	printf("pointer->data = %d\n", pointer->data);
}
```

Linked List code

```
typedef struct node {
    int data;
    struct node * next;
} node_t;

node_t* head = null;

int main() {
        insert(4);
        insert(5);
        insert(6);
        insert(20);
        insert(30);
        print_all();
}
```

Insert

```
void insert(int data){

	node_t *new_node,*current_node,*previous_node;
	current_node = head;
	new_node = (node_t*)malloc(sizeof(node_t));
	new_node->data = data;
	new_node->next = NULL;
	if(head != NULL){
		while(current_node){
			previous_node = current_node;
			current_node = current_node->next;
		}
		previous_node->next = new_node;
		new_node->next = NULL;
	}else{
		head = new_node;
	}
}
```

print_all

```
void print_all(){
	node_t * current_node;
	current_node = head;
	while(current_node){
		printf("%d -> ",current_node->data);
		current_node = current_node->next;
	}
	printf("NULL\n");
}
```

Get

```
node_t * get(int data){
	node_t * current_node, *returned_node;
	current_node = head;
	returned_node = NULL;
	while(current_node){
		if(current_node->data == data){
			returned_node = current_node;
			break;
		}
		current_node = current_node->next;
	}
	return returned_node;
}
```

Insert improvement

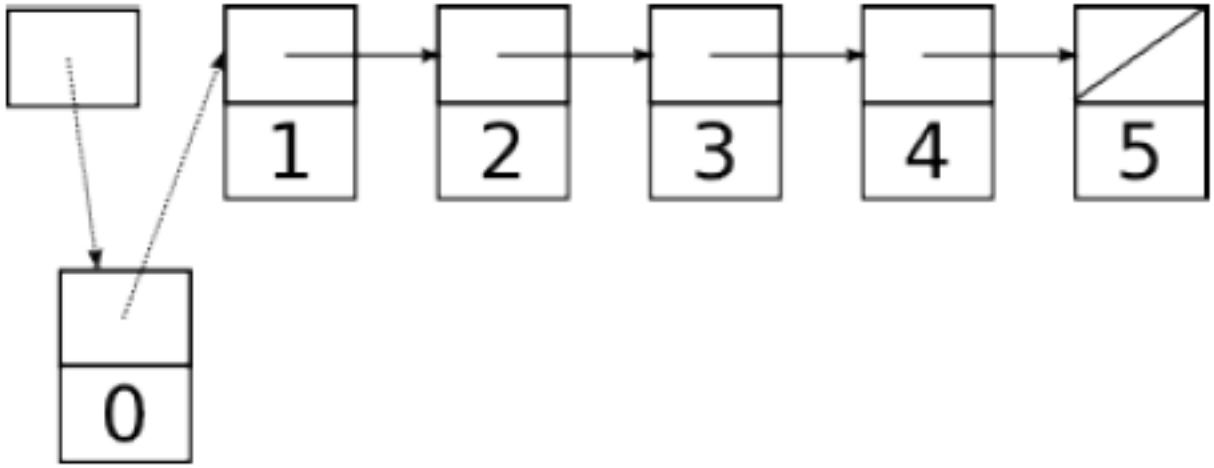
- Insert ใน Linked List ใช้เวลานานหากมีสมาชิกใน Linked List เยอะ
- มีใครสามารถเสนอแนวทางในการแก้ปัญหา?
- ใช้ tail ในการชี้ไปยังสมาชิกตัวสุดท้าย

Let’s see animation

- http://visualgo.net/list.html

Stack

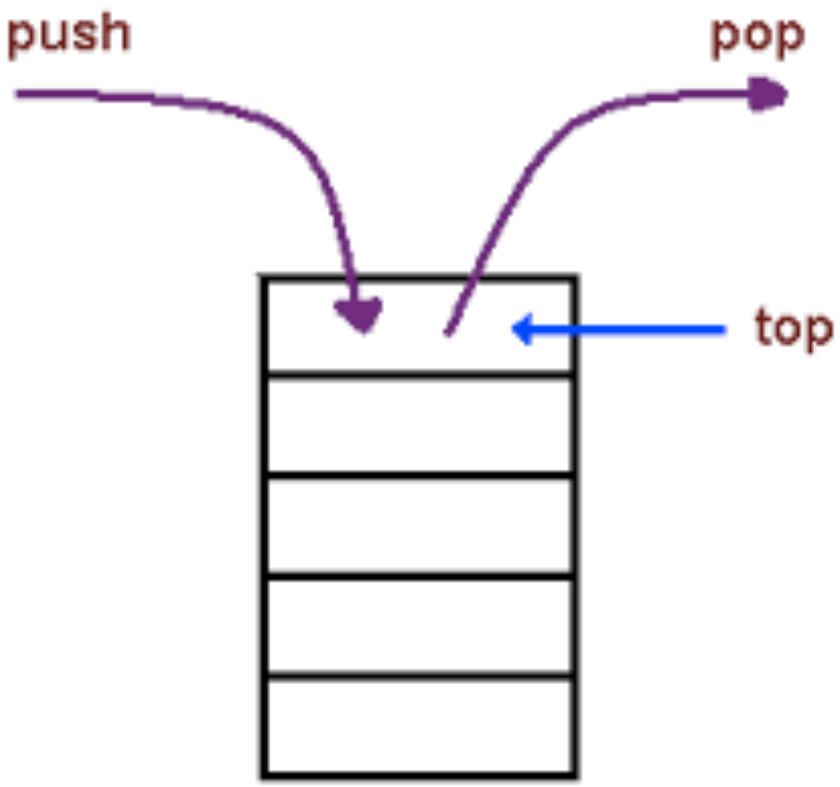
- Stack เป็น data structure รูปแบบหนึ่งที่ใช้ในการเก็บข้อมูล
- ลักษณะของ stack จะเป็นแบบ First-in, last-out



Stack function

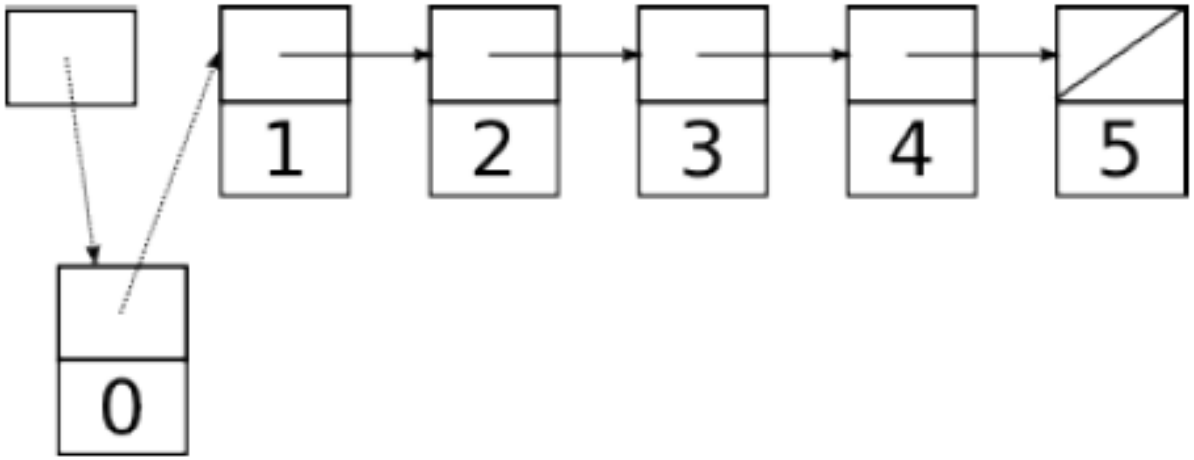
- Function ใน stack นั้นจะมีชื่อเรียกที่แตกต่างออกไปแต่โดย concept แล้วมีลักษณะคล้ายกับ Linked List
- ฟังก์ชันใน stack จะประกอบไปด้วย
  - Push ใส่ข้อมูลไปยังชั้นบนสุดของ stack
  - Pop ดึงข้อมูลชั้นบนสุดของ stack ออกมา (ข้อมูลจะหายไปจาก data structure)
  - Top อ่านข้อมูลที่อยู่บนสุด

Stack



การ implement Stack บน Linked List

- การนำ Linked List มาเป็นโครงสร้างพื้นฐานของ stack นั้นสามารถทำได้ดังรูปด้านล่าง
- เมื่อ push ก็เปลี่ยนเอา head ชี้ไปที่ node ใหม่ (node 0)
- เมื่อ pop ก็นำ head ชี้ไปที่ node ที่ 1 แล้วนำเอา node สุดออกมา



Stack push

```
void push(int data){
	node_t* new_node = (node_t*) malloc(sizeof(node_t));
	new_node->data = data;
	new_node->next = head;
	head = new_node;
}
```

Stack pop & top

```
node_t* pop(){
        node_t* returned_node;
        returned_node = head;
        if(head){
                head = head->next;
        }
        return returned_node;
}

int top(){
        return head->data;
}
```

Stack

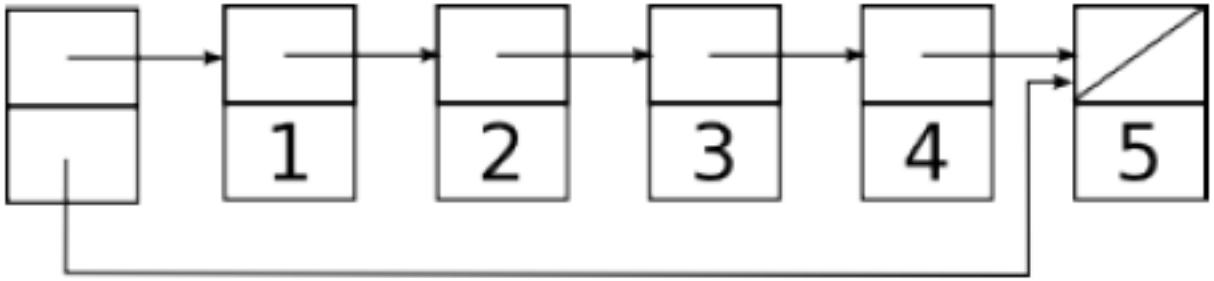
- Push ความเร็ว = 1
- Pop ความเร็ว = 1
- Top ความเร็ว = 1
- Search?

Application of stack

- โปรแกรม reverse คำ
- Depth First Search

Queue

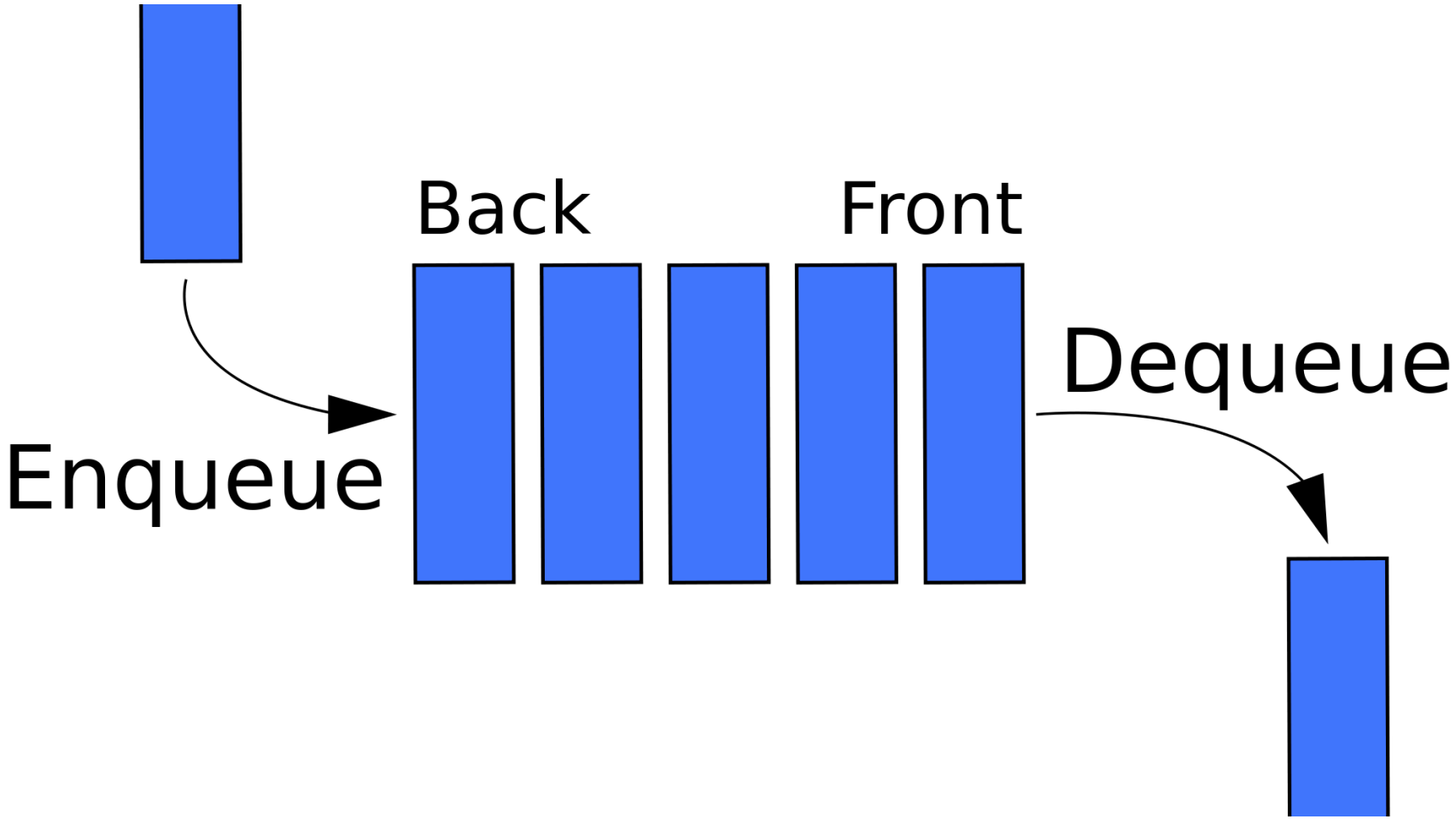
- Queue เป็น data structure อีกรูปแบบหนึ่งที่ใช้ในการเก็บข้อมูล
- ลักษณะของ queue นั้นจะเป็นแบบ First-in, first-out



Queue functions

- Queue จะประกอบไปด้วย function ที่ทำหน้าที่ในการใส่ข้อมูลใหม่ และ ดึงข้อมูลที่อยู่ใน queue ออกมา
  - Enqueue ใส่ข้อมูลลงใน queue
  - Dequeue ดึงข้อมูลที่อยู่ใน queue ออกมา

Queue



Queue enqueue

```
void enqueue(int data){
	node_t *new_node,*current_node,*previous_node;
	current_node = head;
	new_node = (node_t*)malloc(sizeof(node_t));
	new_node->data = data;
	new_node->next = NULL;
	if(head != NULL){
		while(current_node){
			previous_node = current_node;
			current_node = current_node->next;
		}
		previous_node->next = new_node;
		new_node->next = NULL;
	}else{
		head = new_node;
	}
}
```

Queue dequeue

```
int dequeue(){
	int returned_data = 0;
	node_t * temp_node;
	temp_node = head;
	if(temp_node){
		head = temp_node->next;
		returned_data = temp_node->data;
		free(temp_node);
	}
	return returned_data;
}
```

Queue ความเร็ว

- Enqueue ความเร็วเฉลี่ย n
- Dequeue ความเร็วเฉลี่ย 1

Application of queue

- โปรแกรมแจกบัตรคิว
- Breath First Search

Searching is the problem

- การหาข้อมูลใน Linked List, Stack, Queue ใช้เวลานานหากมีจำนวนสมาชิกเยอะ
- โปรแกรมบางชนิดต้องการการค้นหาที่รวดเร็ว
- ต้องการ Data Structure ที่ให้ความเร็วในการ search = 1
- Array เป็นตัวอย่างของ data structure ที่ความเร็วในการค้นหา = 1 แต่มีข้อจำกัดตรงที่ index และข้อมูลไม่มีความสัมพันธ์กัน

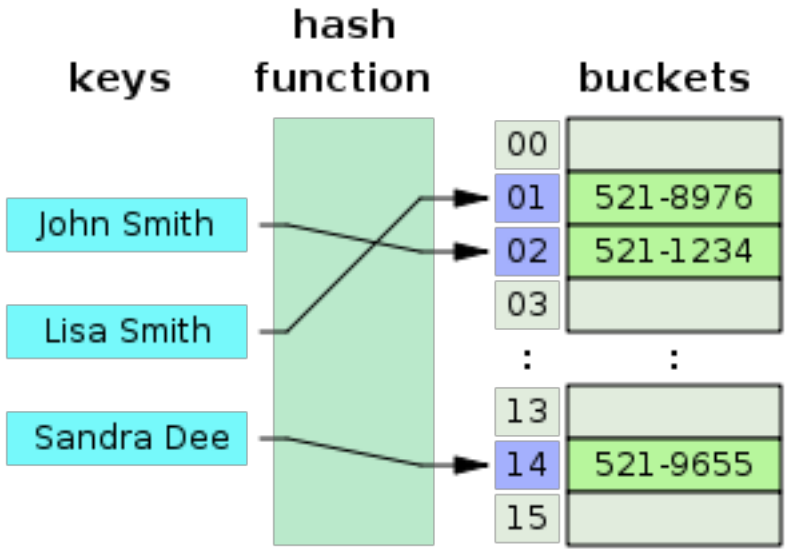
Hash Table

- Hash table เป็น data structure ที่ถูกสร้างขึ้นเพื่อให้
  - Searching มีความเร็ว = 1
  - Inserting มีความเร็ว = 1
  - Delete มีความเร็ว = 1
- การทำงานของ Hash Table มีลักษณะเป็น key/value mapping

```
title = {} # empty dictionary
title["Barack" ] = "President"
user = "Barack"
print ("Welcome"  + title[user] + " "  + user)
```

ลักษณะของ Hash Table data structure

- Hash Function เป็น function ที่ใช้ในการสร้าง key จากข้อมูล
  - Hash function ทำการ generate random number จาก string
- Hash table จะมี slot จำกัดซึ่งจองพื้นที่ไว้ล่วงหน้า



ตัวอย่างการใช้งาน hash table ที่มีขนาด 4 ช่อง

สมมติต้องการเก็บข้อมูล String “Hello World” ลงใน hash table

1. ทำการนำ “Hello World” เข้า hash function
   hash(“Hello World”) = 00
2. จะเห็นว่า hash function return ค่าออกมาเป็น 00
   ดังนั้น จะเก็บ hello world ไว้ที่ช่อง 00
3. เมื่อต้องการ ค้นหา “Hello World” ก็นำเอาเข้า hash function
   hash(“Hello World”) = 00
4. Hash function จะทำการคืนค่าเดิมเสมอ คือ 00
   ดังนั้นโปรแกรมจึงสามารถเข้าถึงข้อมูลได้อย่างรวดเร็ว

00	**Hello World**
01	
10	
11	

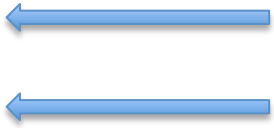
Hash collision

- เนื่องจาก hash function จะ return ค่าเป็น random string ที่อยู่ใน set ที่จำกัด
  - จากตัวอย่าง random string มีได้แค่ 00 01 10 11
- ในบางกรณี hash function อาจจะ return ค่าออกมาเป็นค่าเดียวกัน
  - เช่น hash(“Hello World”) และ hash(“This is me”) อาจจะมีค่าเป็น 00 เหมือนกัน

Open addressing

- Open addressing เป็นหนึ่งในวิธีการแก้ไขปัญหา hash collision
- Open addressing จะเก็บข้อมูลลงใน slot ถัดไปที่ว่างอยู่หากเกิดการชนกันของ address

00	**Hello World**
01	This is me
10	
11	



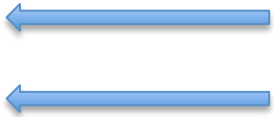
hash(“This is me”) = 00
collision detect

Store “This is me” in 01

Retrieve data

- ในกรณีที่มีการใช้ open addressing ในการแก้ไขปัญหา hash collision เมื่อต้องการทำงานเรียกดูข้อมูล ก็ต้องตรวจสอบข้อมูลก่อน

00	**Hello World**
01	This is me
10	
11	



hash(“This is me”) = 00
เนื่องจาก ข้อมูลในช่องที่ 00 ไม่ตรงกับ “This is me” ดังนั้นต้องไล่หาไปอีกช่อง จึงพบกับ “This is me” ถูกเก็บอยู่ใน 01

ภาษา C และ Hash table

- ภาษา C ไม่มี hash table ให้เรียกใช้ ดังนั้น ผู้พัฒนาต้องเขียน hash table ขึ้นมาเอง
- หัวใจของการเขียน hash table คือ
  - การเขียน hash function ที่ทำให้เกิดการ collision ให้น้อยที่สุด (ข้อมูลกระจายมากที่สุด)