

Linked List

Assembled for 204112
by Kittipitch Kuptavanich
Ratsameetip Wita

Dynamic Data Structure

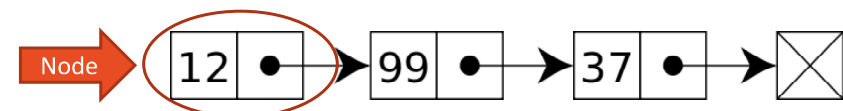
- ไม่จำเป็นต้องมีการระบุจำนวนข้อมูลที่ต้องการเก็บไว้ล่วงหน้า
- จองที่ในหน่วยความจำ เมื่อมีการเพิ่มข้อมูล
 - จองหน่วยความจำแยกกันได้เป็นหน่วยย่อยๆ
- ปล่อยหน่วยความจำที่ไม่ต้องการใช้ เมื่อมีการลบข้อมูล

Static Data Structure

- พิจารณา การเก็บข้อมูลแบบ Array
 - มีขนาดจำกัด ต้องระบุไว้ล่วงหน้าว่า Array มีความจุเท่าไร
 - จำเป็นต้องจองพื้นที่เก็บข้อมูลที่ต่อเนื่องกันในหน่วยความจำเป็นชิ้นเดียว
 - มีความยุ่งยากในกรณีต่อไปนี้
 - กรณีที่เก็บข้อมูลจนเต็มพื้นที่
 - ต้องการลดขนาดเนื่องจากจองที่ไว้เกินกว่าที่ต้องการมาก

Linked List

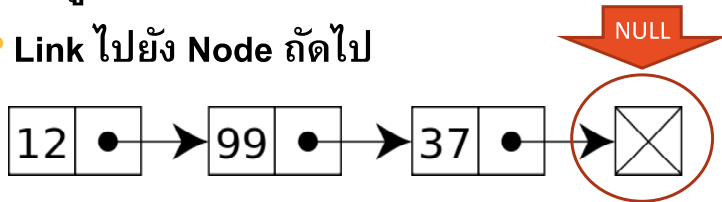
- Definition: Linked List (ลิงก์ ลิสต์) เป็น Dynamic Data Structure ที่มีข้อมูล (record) เรียงต่อกัน โดยแต่ละ element มี link เชื่อมไปยัง record ถัดไป
- Linked List มีได้หลายประเภท เช่น Singly Linked, Doubly Linked, Circular – ใน class นี้เราจะพิจารณาแค่ Singly Linked List



- เรียกแต่ละ element ใน linked list ว่า Node

Linked List [2]

- แต่ละ Node จะประกอบด้วย
 - ข้อมูลที่ต้องการเก็บ
 - Link ไปยัง Node ถัดไป



- ตำแหน่งท้ายสุดของ List จะมี Link ชี้ไปที่ **NULL**
- เราจะเข้าถึงข้อมูลใน List โดยการเก็บตำแหน่งแรกสุดของ List (head) ไว้

Syntax – Node Declaration

```
struct node
{
    int data;          /* can be any data type */
    struct node *next;
};
struct node *head;    ← ตำแหน่งหัว List (head)
```

```
/* สามารถใช้ typedef เพื่อสร้าง alias (shortcut) */
typedef struct node nodeT;
```

Dynamic Memory Allocation

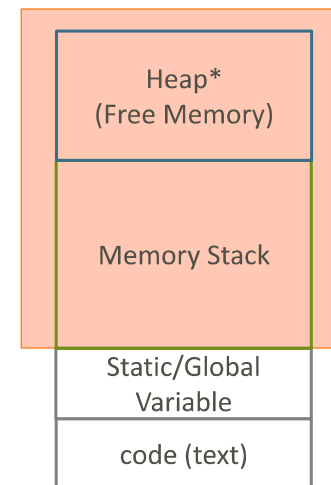
- โดยปกติแล้ว ตัวแปรใด ๆ ที่สร้างขึ้นใน function จะไม่สามารถถูกเข้าถึง (เขียน/อ่าน) ได้ เมื่อจบการทำงาน function นั้น ๆ

```
pointT *createPoint(int x, int y ) {
    pointT newPoint = {x,y};
    return &newPoint;
}
```

Will **NOT** Work!!

Application Memory Allocation

- ในการรันโปรแกรม จะมีการจัดสรรหน่วยความจำ 2 รูปแบบคือ Stack และ Heap
- Stack จัดการโดย OS
 - เก็บค่าตัวแปร แต่ละฟังก์ชัน
 - คืนพื้นที่อัตโนมัติ หลังจบฟังก์ชัน
- Heap จัดการจองและคืนโดยโปรแกรมเมอร์ ผ่านคำสั่ง malloc(), calloc(), realloc() และ free()



Dynamic Memory Allocation [2]

- **Stack**
 - ข้อมูลที่มีขนาดเล็ก ใช้เฉพาะในฟังก์ชันใดฟังก์ชันหนึ่ง
 - จัดการให้อัตโนมัติ ไม่จำเป็นต้องเขียนคำสั่งเพิ่มเติม
- **Heap** เหมาะกับ
 - การเก็บข้อมูลอาเรย์ หรือ struct ขนาดใหญ่ เช่น `int arr[1000000];`
 - ตัวแปรที่ต้องการเก็บไว้ใช้ร่วมกันหลายฟังก์ชัน
 - อาเรย์หรือ struct ที่มีการเพิ่มหรือลดขนาดได้
 - โครงสร้างข้อมูลแบบต่าง ๆ (เรียนในวิชา Data Structure)
 - Programmer จัดการการใช้งานเอง

Memory Deallocation in Stack

```
#include <stdio.h>
int square(int i){
    return i*i;
}

int squareOfSum(int x,int y){
    int z= square(x+y);
    return z;
}

int main(){
    int a=4, b=8;
    int total;
    total = squareOfSum(a,b);
    printf("output = %d",total);
    return 0;
}
```

เมื่อ `square()` ทำงานถึงคำสั่ง `return` โปรแกรมจะคืนพื้นที่สำหรับฟังก์ชัน ให้กับระบบโดยอัตโนมัติ และคืนค่าตัวแปรไปยังฟังก์ชันที่เรียกใช้งาน

เมื่อ `squareOfSum()` จบการทำงาน ระบบจะทำการคืนพื้นที่ใน Stack ให้กับ `main()`

พื้นที่ของ `main()` จะถูกจองไว้จนโปรแกรมจบการทำงาน

Memory Allocation in Stack

```
#include <stdio.h>
int square(int i){
    return i*i;
}

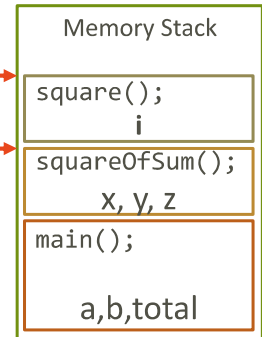
int squareOfSum(int x,int y){
    int z= square(x+y);
    return z;
}

int main(){
    int a=4, b=8;
    int total;
    total = squareOfSum(a,b);
    printf("output = %d",total);
    return 0;
}
```

ฟังก์ชัน `squareOfSum()`; ทำการเรียกฟังก์ชัน `square(x+y)`; ระบบจะทำการจองพื้นที่ใน stack เพิ่มเติมให้กับฟังก์ชัน `square()`;

เมื่อเรียกคำสั่ง `squareOfSum(a,b)`; ระบบจะทำการจองพื้นที่ไว้ใน Stack ต่อจาก `main()` ที่จองไว้ก่อนหน้านี้

เมื่อโปรแกรมเริ่มทำงาน ระบบจะทำการจองพื้นที่เพื่อเก็บตัวแปรสำหรับ `main()` ไว้ใน Stack



Dynamic Memory Allocation [3]

หากต้องการสร้างตัวแปรที่มี data type เป็น struct ขึ้น โดยที่ต้องการให้มีการเข้าถึงข้อมูลในตัวแปรนั้น ๆ ได้ นอก function ที่สร้างตัวแปรนั้น ๆ

- จำเป็นต้องมีการจองหน่วยความจำไว้อย่างถาวร โดยใช้ฟังก์ชัน `malloc()`

void *malloc(size_t n)

- `malloc` = memory allocate
- Return pointer (ไม่ระบุชนิด - `void *`) ไปยังหน่วยความจำที่จองไว้ตามขนาดที่ระบุ (`n`)
- หากจองหน่วยความจำไม่สำเร็จ จะ return `NULL`

Dynamic Memory Allocation [4]

```
nodeT *newNode = (nodeT *) malloc(sizeof(nodeT));
```

- เนื่องจาก malloc() มีการ return ค่าเป็น **void *** เราจำเป็นต้องมีการเปลี่ยนชนิด (type casting) ของ pointer เป็นชนิดที่ต้องการ
- ฟังก์ชัน sizeof() ใช้เพื่อวัดขนาดของ หน่วยความจำที่ต้องการจอง ในกรณีนี้เราต้องการขนาดเท่ากับขนาดของหนึ่ง **nodeT**
- หากต้องการจองสำหรับ **int** array ขนาด **10** หน่วย

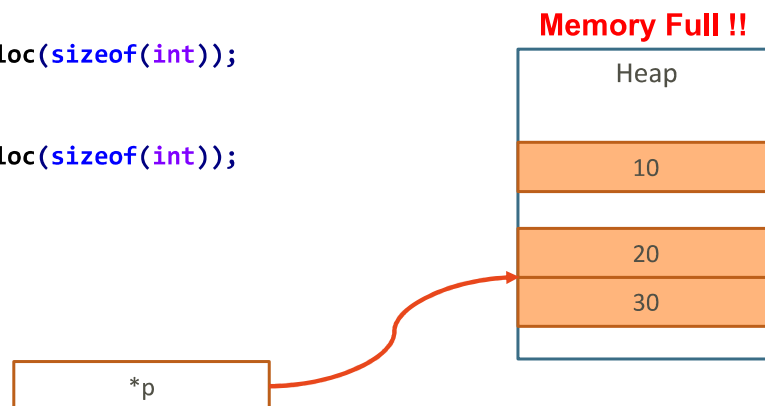
```
malloc(sizeof(int) * 10)
```

Dynamic Memory Allocation [6]

```
int main()
{
    int i;
    int *p= malloc(sizeof(int));
    *p=10;

    p= malloc(sizeof(int));
    *p=20;

    p= malloc(sizeof(int));
    *p=30;
}
```



Dynamic Memory Allocation [5]

```
void *calloc(size_t n, size_t size)
```

- จองพื้นที่หน่วยความจำในลักษณะ array ที่มีสมาชิก **n** ตัว โดยแต่ละตัวมีขนาด **size** byte
- พร้อมกำหนดให้ค่าเริ่มต้นทั้งหมดของสมาชิกแต่ละตัวเป็น 0

ข้อควรระวัง: หากมีการจองหน่วยความจำไว้อย่างถาวรเพิ่มขึ้นเรื่อยๆ หน่วยความจำของคอมพิวเตอร์ก็จะเต็มในที่สุด

Dynamic Memory Allocation [7]

ดังนั้นเมื่อไม่ใช้ตัวแปรใด ๆ ที่มีการจองหน่วยความจำไว้แบบถาวร เราจำเป็นต้องคืนหน่วยความจำนั้น ๆ โดยการใช้ฟังก์ชัน **free()**

```
void free (void *)
```

```
/* for example */
free(newNode)
```

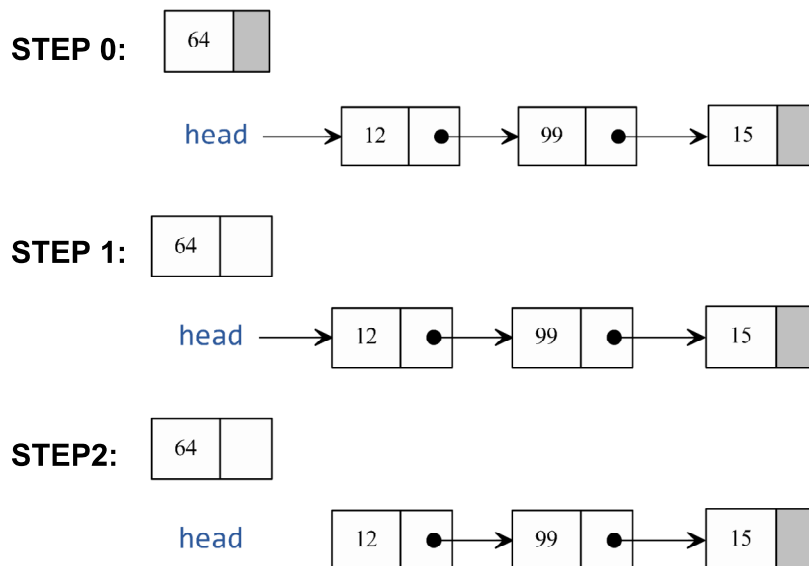
- ใน application ใด ๆ จำนวน memory ที่จองด้วย **malloc()** จะต้องเท่ากับ memory ที่ **free()** ออก ในที่สุด เพื่อป้องกัน memory leak

Creating a Node

```
nodeT *createNode (int data) {
    nodeT *newNode = malloc(sizeof(nodeT));
    if (newNode == NULL) /* not enough memory */
        return NULL;

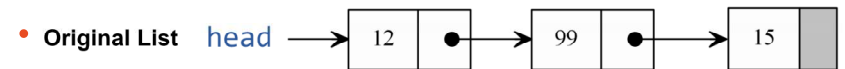
    newNode->data = data; /*same as (*newNode).data*/
    newNode->next = NULL;
    return newNode;
}
```

Add Node to Front

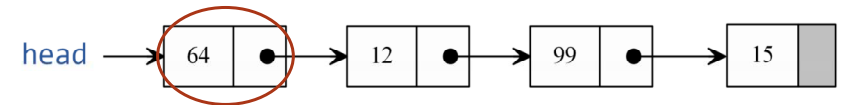


Adding New Node to a List

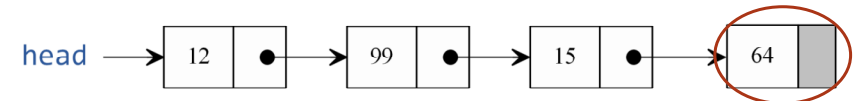
- เราสามารถเพิ่ม Node ใหม่เข้าไปที่ส่วนหัว (add to front) หรือส่วนท้าย (add to back)



- Add to Front



- Add to Back



ถ้า head เป็น NULL
การ add newNode จะเป็นการสร้าง List ใหม่

Add Node to Front [2]

```
nodeT *addToFront(nodeT *head, int data)
{
    nodeT *newNode = createNode(data);
    if (newNode == NULL) /* not enough memory */
        return head;
    newNode->next = head;
    return newNode;
}
```

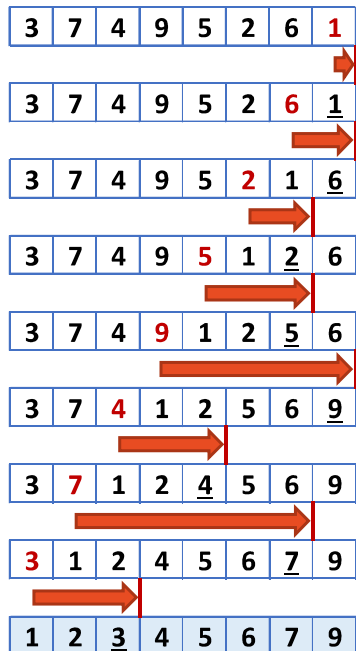
ถ้าต้องการ add to back จะต้องทำอย่างไร?

List Iterating

- การเข้าถึงทีละสมาชิกสามารถทำได้โดย

```
for (p = head; p != NULL; p = p->next) {
    /* do something */
}

for (p = head; p->next != NULL; p = p->next) {
    /* do something */
}
```



Insert in Order

Insertion Sort (Revisited)

- เริ่มจากตำแหน่งขวาสุด
- จับวิ่งไปทางขวาจนพบค่ามากกว่า
จึงหยุด

List Iterating [2]

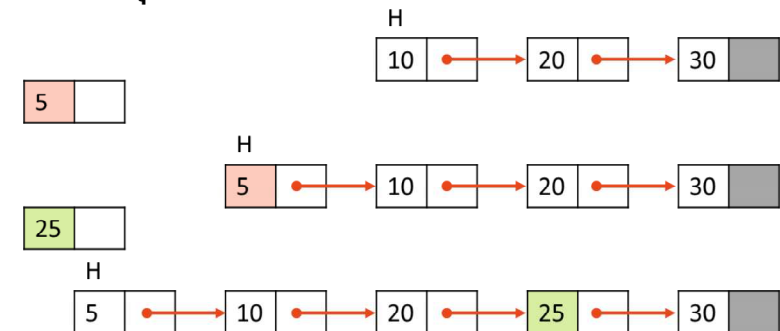
- หรือใช้ `while` เช่น

```
void printList(nodeT *head)
{
    nodeT *currNode = head;
    while (currNode != NULL) {
        printf("%d -> ", currNode->data);
        currNode = currNode->next;
    }
    printf("NULL\n");
}
```

Insert in Order [2]

- ทำนองเดียวกันกับ insertion sort

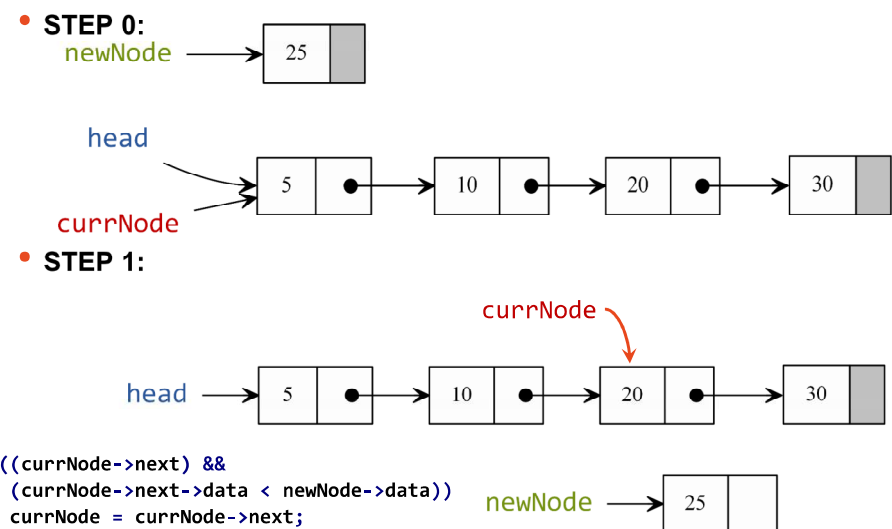
- จับ `newNode` วิ่งไปทางขวา (วิ่งจาก head ไปทางท้าย list) และหยุดเมื่อเจอ node ที่มีค่ามากกว่า



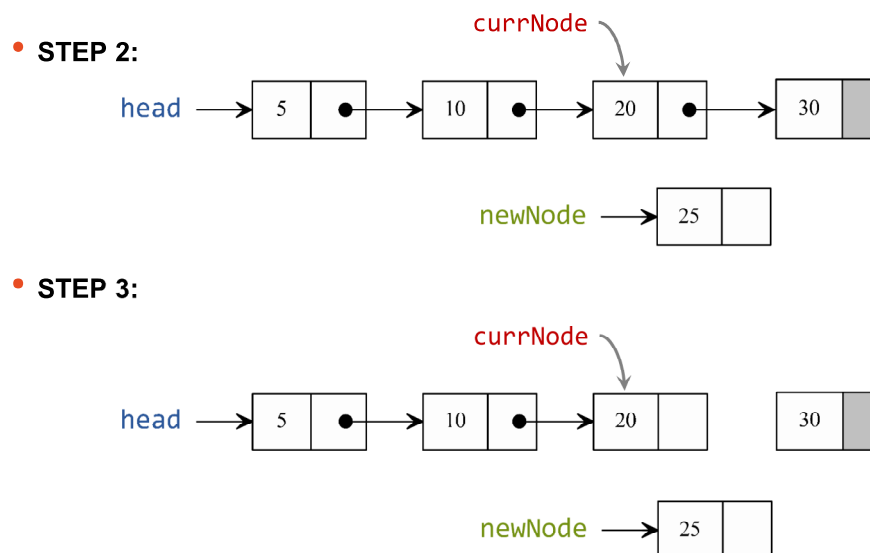
Insert in Order [3]

1. **Case 1:** หาก `newNode->data` น้อยกว่า `head->data` หรือ `head == NULL`
 - `return addToFront(head, data)`
2. **Case 2:** วิ่งไล่ที่ ละ Node จนเจอ `nodeX` ที่ `nodeX->data > newNode->data`
 - นำ `newNode` ไว้ที่ตำแหน่งหน้า `nodeX` นั้น

Insert in Order [4]



Insert in Order [5]



Find in List

```
nodeT *findInList(nodeT *head, int key)
{
    nodeT* currNode = head;
    while (currNode && currNode->data < key) {
        currNode = currNode->next;
    }
    if (
        return
    )
}
```

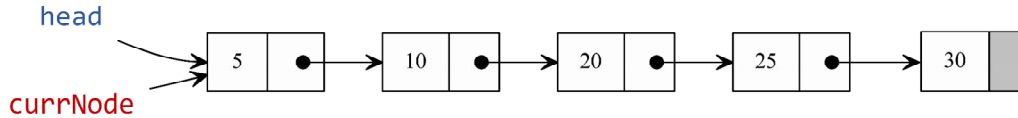
Delete from List

- Case 1: key อยู่ ตำแหน่ง head (Easy)
- Case 2: key อยู่ภายใน list

STEP 0:

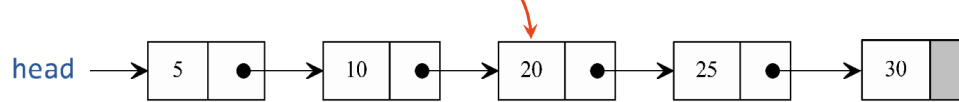
key = 25

Case 2



STEP 1:

currNode



```
while ((currNode->next) &&
      (currNode->next->data < key))
    currNode = currNode->next;
```

A First Book of ANSI C, 4th Edition

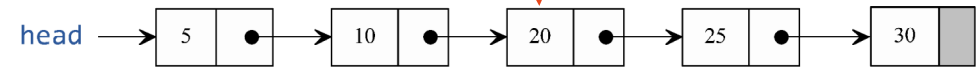
33

Delete from List [2]

```
if (currNode->next && currNode->next->data != key) {
    printf("Key %d not found\n", key);
    return head;
}
```

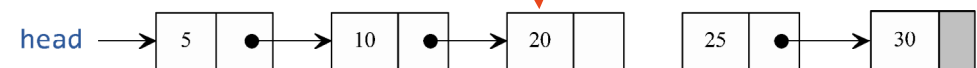
STEP 2:

currNode



STEP 3:

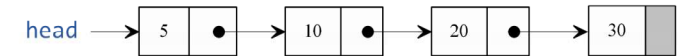
currNode



STEP 4:

free(

)



A First Book of ANSI C, 4th Edition

34

Array vs Linked List

- กรณีต้องการ delete
 - Array: หลังจาก delete ต้องขยับสมาชิกอื่น ๆ เพื่อมาแทนที่ว่าง
 - Linked List: ไม่มีผลกระทบหลังการ delete
- กรณีต้องการ insert
 - Array: ต้องขยับสมาชิกอื่น ๆ เพื่อสร้างที่ว่างให้ insert ได้
 - Linked List: insert ได้โดยไม่ต้องขยับสมาชิกอื่น ๆ

A First Book of ANSI C, 4th Edition

37

Other Types of Data Structures

Arrays

Lists

Trees

Binary trees

B-trees

Heaps

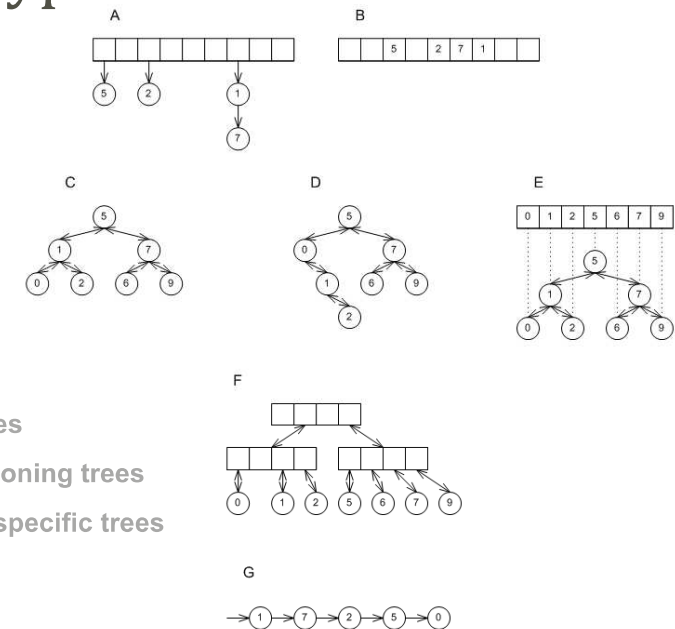
Tries

Multway trees

Space-partitioning trees

Application-specific trees

Graphs



A First Book of ANSI C, 4th Edition

38

Reference

- <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-087-practical-programming-in-c-january-iap-2010/>
- http://en.wikipedia.org/wiki/List_of_data_structures