Lecture 09

외부 라이브러리

리뷰

- **void** 포인터
  - 어떤 형의 포인터로 변환될 수 있음

```
int x; void *px = &x;
float f; px = &f;
```

  - * 연산자를 바로 할 수 없으며, 다른 형의 포인터로 변환하고 나서 * 연산자를 할 수 있음

```
int x; int *px = &x;
int y = *px; // 에러
int z = *(int*)px; // 정상
```

  - 원래 C에서 **void** 포인터에 연산을 할 수 없지만, GNU C에서는 **void** 포인터에 연산을 할 수 있음

```
int arr[] = {1, 2};
void *p = arr;
p = p+sizeof(int);
```

리뷰

- 함수의 포인터
  - 함수는 변수가 아니지만, 메모리에 저장되므로 메모리 번지는 할당됨
    → 함수의 포인터를 선언할 수 있음

```
int (*fp)(int);
int (*fp)(void*);
```

  - 함수의 포인터는 다른 함수로 전달될 수 있음

```
int func(int, int, int (*fp)(void*));
```

  - 함수의 포인터로 이루어지는 배열은 선언될 수 있음

```
int (*farr[])(int) = {func1, func2, func3};
```

리뷰

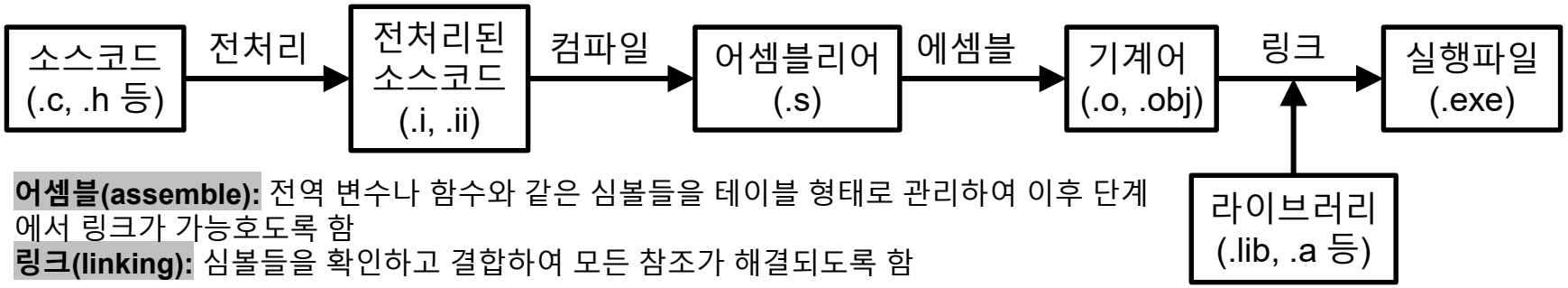
- 해시 테이블
  - 데이터를 효율적으로 저장하고 접속하기 위한 연결리스트로 이루어지는 배열

```
typedef struct wordrec {
    char *word;
    struct wordrec *next;
} wordrec;
wordrec *table[1000];
```

  - 배열의 각 요소에 인덱스를 알아낼 수 있는 키를 할당함
  - 키-인덱스 변환을 해시 함수가 담당함
  - 서로 다른 키를 갖고 있지만 같은 인덱스로 매핑될 수 있으며, 이를 충돌이라고 부름

라이브러리

- 다음과 같은 작업을 위한 함수들을 제공함
  - 매크로
  - 형 정의
  - 문자열 처리
  - 수학적 연산
  - …
- 컴파일 과정의 **링크**(linking) 단계에서 **심볼**(symbol)을 통해 라이브러리에서 제공된 전역 변수나 함수들은 접속됨



어셈블(assemble): 전역 변수나 함수와 같은 심볼들을 테이블 형태로 관리하여 이후 단계에서 링크가 가능호도록 함
링크(linking): 심볼들을 확인하고 결합하여 모든 참조가 해결되도록 함

라이브러리

- 아래의 프로그램에서는 선언된 전역 변수와 함수들이 무엇인가?

```
#include <stdio.h>
const char msg[] = "hello world";
int main() {
   puts(msg);
    return 0;
}
```

라이브러리

- 소스코드 컴파일

```
gcc -Wall -c hello.c -o hello.o
```

  - 컴파일하여 목적파일(`hello.o`)을 생성함
  - 링크는 아직 진행되지 않으므로 메모리 번지는 할당되지 않음

```
dat@dat-VirtualBox:~/Downloads/test_C_linking$ nm hello.o
0000000000000000 T main
0000000000000000 R msg
                 U puts
```

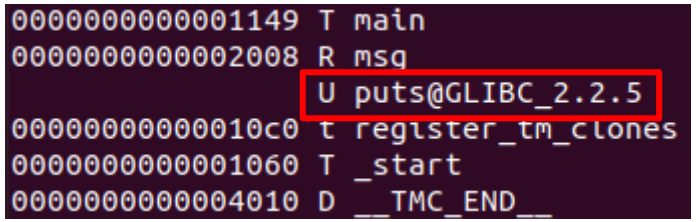
    - `T`는 text를 의미하며 코드와 해당됨
    - `R`는 read-only memory를 의미하며 상수인 전역 변수 `msg`와 해당됨
    - `U`는 undefined symbol을 의미하며 소스 코드에서 정의가 없는 `puts` 함수와 해당됨

라이브러리

- 링크

```
gcc -Wall hello.o -o hello
```

  - 소스 코드에서 정의되어 있는 심볼한테 메모리 번지를 할당함
  - 소스 코드에서 정의되어 있지 않는 심볼들은 라이브러리에 찾아냄



    - `puts`는 `GLIBC_2.2.5`(GNU C 표준 라이브러리) 공유 라이브러리(shared library)에 찾아냈음
    - `puts`와 같은 공유 심볼(shared symbol)의 경우는 링크 단계에서도 메모리 번지는 할당되지 않고 프로그램을 실행할 때만 할당됨
      → **동적 링크**(dynamic linking)라고 부름

전적링크(static linking) 및 동적링크(dynamic linking)

- 프로그램의 전역 변수와 함수들에 메모리 번지는 할당되어야만 프로그램이 실행될 수 있음
- 메모리 번지 할당은 프로그램 컴파일 때도 가능하고, 실행 때도 가능함
  - 전적링크: **컴파일 때** 메모리 번지 할당은 실시됨
    - 장점: 프로그램이 독립적으로 돌릴 수 있음
    - 단점: 프로그램의 크기가 큼
  - 동적링크: 프로그램 **실행 때** 메모리 번지 할당은 실시됨
    - 장점: 프로그램의 크기가 작음
    - 단점: 프로그램 실행 때 공유 라이브러리가 필요함
- `gcc`는 기본적으로 동적링크를 실시함
  - 전적링크를 원한다면 `-static` 플래그를 사용함

전적링크

- 전적링킹 플래그 사용

```
gcc -Wall -static hello.o -o hello_static
```

  - `puts` 함수에 메모리 번지가 할당됨

```
00000000004152b0 T __pthread_setcancelstate
00000000004152b0 W pthread_setcancelstate
000000000046cfd0 T __pthread_sigmask
000000000046cfd0 W pthread_sigmask
00000000004152e0 T ___pthread_testcancel
00000000004152e0 T __pthread_testcancel
00000000004152e0 W pthread_testcancel
000000000046d0f0 T __pthread_tpp_change_priority
000000000046bca0 T __pthread_tunables_init
0000000000416bb0 t ptmalloc_init.part.0
000000000040c180 W puts
000000000041a8f0 T __pvalloc
000000000041a8f0 W pvalloc
000000000040a770 T qsort
000000000040a390 T __qsort_r
000000000040a390 W qsort_r
```

전적링크

- 실행 파일의 크기 비교
  - 동적링크: `hello(16K)`
  - 전적링크: `hello_static(880K)`

```
dat@dat-VirtualBox:~/Downloads/test_C_linking$ ls -list -sh
total 904K
3672203 880K -rwxrwxr-x 1 dat dat 880K  8월 21 16:57 hello_static
3672202  16K -rwxrwxr-x 1 dat dat  16K  8월 21 16:56 hello
3672195 4.0K -rw-rw-r-- 1 dat dat 1.5K  8월 21 16:56 hello.o
3672199 4.0K -rw-rw-r-- 1 dat dat   94  8월 21 16:55 hello.c
```

- 프로그램 실행 때 필요한 공유 라이브러리를 찾기

`objdump -p hello | grep NEEDED`

```
dat@dat-VirtualBox:~/Downloads/test_C_linking$ objdump -p hello | grep NEEDED
  NEEDED               libc.so.6
dat@dat-VirtualBox:~/Downloads/test_C_linking$ objdump -p hello_static | grep NEEDED
```

외부 라이브러리 링크

- 컴파일 과정에서 표준 라이브러리는 기본적으로 링크됨
- 외부 라이브러리를 링크하려면 플래그를 사용해 야 함
  - 공유 라이브러리 확장자: `lib`*`name`*`.so` (Linux), `lib`*`name`*`.dll` (Windows)
  - 전적(static) 라이브러리 확장자: `lib`*`name`*`.a`
  - 컴파일 때 사용하는 플래그: `-l`*`name`*

mymath.c

```
int add(int a, int b) {
   return a+b;
}
```

main.c

```
int add(int,int);
int main() {
   ...
   int a=10, b=15;
   printf("%d\n",add(a,b));
   ...
}
```

라이브러리 컴파일

```
gcc -c mymath.c -o mymath.o
ar rcs libmymath.a mymath.o (전적 라이브러리)
```

동적 라이브러리

```
gcc -shared -fPIC -o libmymath.so mymath.o (Linux)
gcc -shared -fPIC -o libmymath.dll mymath.o (Windows)
```

main 컴파일

```
gcc -Wall main.c -lmymath -L. -static -o main (전적 링크)
gcc -Wall main.c -lmymath -L. -o main (동적 링크)
```

실행파일

main.exe

외부 라이브러리 링크

- 외부 라이브러리 경로
  - 표준 라이브러리 경로와 동일한 경우 아무 조치도 취하지 않음
  - 표준 라이브러리 경로와 다른 경우 `-L` 컴파일 플래그를 사용해 야 함

```
gcc -Wall main.c -lmylib -L mylib경로 -o main
```

① **현 폴더**
```
| main.c
| mylib.a
```

② **현 폴더**
```
| main.c
| lib
  | mylib.a
```

③ **lib**
```
| mylib.a
| 현 폴더
  | main.c
```

```
① gcc -Wall main.c -lmylib -L. -o main
② gcc -Wall main.c -lmylib -L ./lib -o main
③ gcc -Wall main.c -lmylib -L ../lib -o main
```

외부 라이브러리 링크

- 공유 라이브러리와 동적링크
  - 공유 라이브러리를 만들기
    - Linux : `gcc -shared -fPIC -o mylib.so mylib1.o mylib2.o`
    - Windows : `gcc -shared -fPIC -o mylib.dll mylib1.o mylib2.o`
  - 컴파일: `gcc -Wall main.c -lmylib -L mylib`경로 `-o main`
  - 프로그램을 실행하기 전 공유 라이브러리 경로를 지정해 야 함
    - Linux : `export LD_LIBRARY_PATH=mylib`경로
    - Windows
      - 프로그램과 같은 폴더에 있을 경우 아무 조치도 취하지 않음
      - 그 외의 경우, `PATH` 환경 변수에 공유 라이브러리 경로를 추가함

심볼 해결(symbol resolution)

- 사용자가 `puts` 함수를 직접 구현하고 코드에서 사용하고 싶다면 어떻게 하면 되는가?
- 프로그램 컴파일 때 심볼은 불러오는 순서대로 해결됨
  - 사용자가 구현한 `puts` 함수는 `libmyputs.a` 혹은 `libmyputs.so` 라이브러리에 있다고 가정함
  - 어떤 `puts` 함수가 불러오게 될 지 프로그램 컴파일 때 라이브러리를 지정해주는 순서에 따름
    - `gcc -Wall hello.c -lmyputs -L. -o hello`
    → 사용자가 구현 `puts` 함수 불러오게 됨
    - `gcc -Wall hello.c -lc -o hello`
    → 표준 라이브러리에서 있는 `puts` 함수가 불러오게 됨
    - `gcc -Wall hello.c -lc -lmyputs -L. -o hello`
    → 표준 라이브러리에서 있는 `puts` 함수가 불러오게 됨
    - `gcc -Wall hello.c -lmyputs -L. -lc -o hello`
    → 사용자가 구현 `puts` 함수 불러오게 됨

심볼 해결(symbol resolution)

- 사용자가 구현한 puts 함수의 예

myputs.c

```
#include <stdio.h> // putchar 함수를 사용하기 위함

int puts(const char* str) {
    if (str == NULL)
        return EOF;

    putchar('*'); // 표준 puts 함수와 차별하기 위함

    while (*str != '\0') {
        if (putchar(*str) == EOF)
            return EOF;
        str++;
    }

    if (putchar('\n') == EOF)
        return EOF;

    return 0;
}
```

라이브러리 컴파일

```
gcc -c myputs.c -o myputs.o
```

전적 라이브러리 만들기

```
ar rcs libmyputs.a -o myputs.o
```

공유 라이브러리 만들기

```
gcc -shared -fPIC -o libmyputs.so myputs.o
```

OpenCV 라이브러리

- 개요
  - OpenCV(Open Source Computer Vision Library)
  - 실시간 영상처리, 컴퓨터 비전 알고리즘 제공
  - Windows, Linux, macOS, iOS, Android 등의 다양한 플랫폼 지원
- API 유형
  - 원래 C API만 지원, 향후 C++, Python, Java, JavaScript API도 지원
  - OpenCV 2.x 이후 C++ API 지원
  - OpenCV 3.x/4.x에서 C API 지원 중단

OpenCV 라이브러리

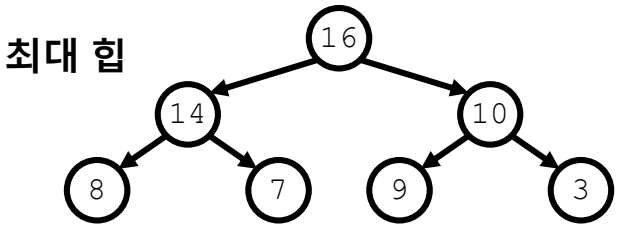
- 예로, 웹캠 캡처 애플리케이션
  - OpenCV 2.4.10 사용
  - 웹캠을 캡처하여 크기 조절, 필터링 등의 영상처리 적용
- 코드 컴파일 시 유의 사항
  - 컴파일 시 사용할 OpenCV 관련 header 파일, 라이브러리의 경로를 지정해야 함

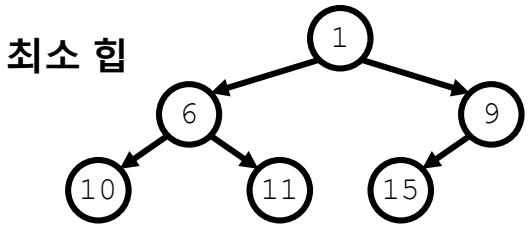
```
gcc main.c -o main
           -I<opencv/build/include 경로>
           -L<opencv/build/x64/vc12/lib 경로>
           -lopencv_core버전 -lopencv_highgui버전 -lopencv_imgproc버전
```

  - 사용할 공유 라이브러리의 경로를 지정해야 함

자료구조 – 힙(heap)

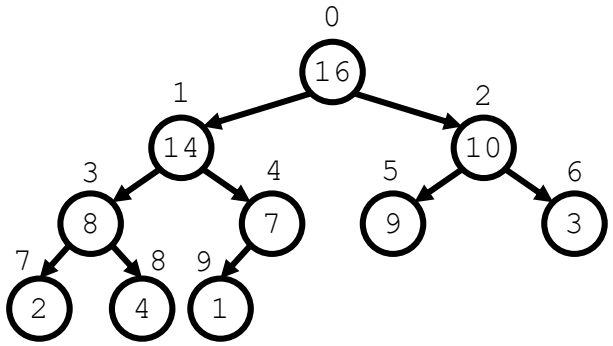
- 힙
  - 완전 이진 트리의 일종으로 최댓값이나 최솟값을 빠르게 찾아내도록 만들어진 자료구조(우선순위 큐라고 부를 수 있음)
  - 느슨한 정렬 상태를 유지함
    - 큰(작은) 값이 상위 노드에 있고 작은(큰) 값이 하위 노드에 있다는 정도
  - 두 개의 종류
    - 최대 힙(max heap)
      - 부모 노드의 값이 자식 노드의 값보다 크거나 같은 완전 이전 트리
    - 최소 힙(min heap)
      - 부모 노드의 값이 자식 노드의 값보다 작거나 같은 완전 이전 트리

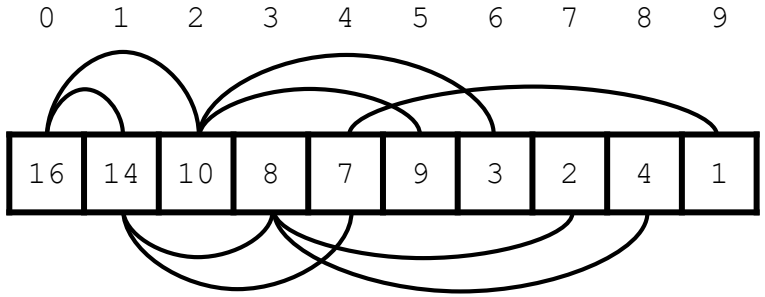




자료구조 – 힙(heap)

- 힙은 일반적으로 배열로 구현됨





  - 부모 노드와 자식 노드의 관계
    - 왼쪽 자식의 인덱스 = (부모의 인덱스) * 2 + 1
    - 오른쪽 자식의 인덱스 = (부모의 인덱스) * 2 + 2
    - 부모의 인덱스 = (자식의 인덱스 – 1) / 2

자료구조 – 힙(heap)

- C에서 힙의 구현

```
#define MAX_ELEMENT 200

typedef struct {
    int key;
} element;

typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} heap;

heap A;
```

자료구조 – 힙(heap)

- 힙의 대표적인 연산
  - `heapify()` : 힙의 특징(최대 힙 혹은 최소 힙)을 유지해줌
  - `build_heap()` : 힙을 만들어줌
  - `insert()` : 힙에 데이터를 추가함
  - `extract()` : 힙에서 최댓(최솟)값을 제거하여 돌려보냄
  - `increase_key()` : 노드의 키 값을 증가시킴

자료구조 – 힙(heap)

- 힙의 특징 유지
  - 최대 힙을 고려함
  - 부모의 인덱스는 `i`라고 가정하면
    - 왼쪽 자식의 인덱스: `left(i) = 2*i+1`
    - 오른쪽 자식의 인덱스: `right(i) = 2*i+2`
  - 자식의 인덱스는 `i`라고 가정하면
    - 부모의 인덱스: `parent(i) = (i-1)/2`
  - **입력:** 힙 `A`와 인덱스 `i`
    - 힙 `A`는 최대 힙의 조건을 만족하지 않을 수 있음
    - 측, 노드 `A.heap[i]`는 자식 노드보다 작을 수 있음
  - **출력:** 힙 `A`는 최대 힙이 되도록 정리해줌

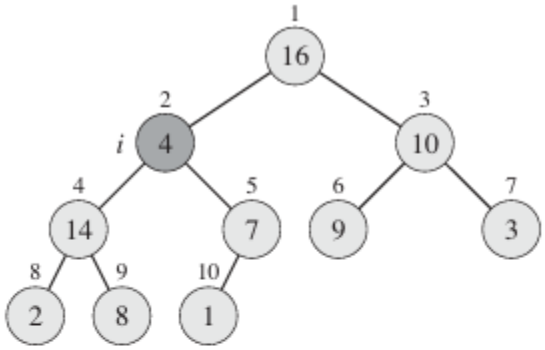
자료구조 – 힙(heap)

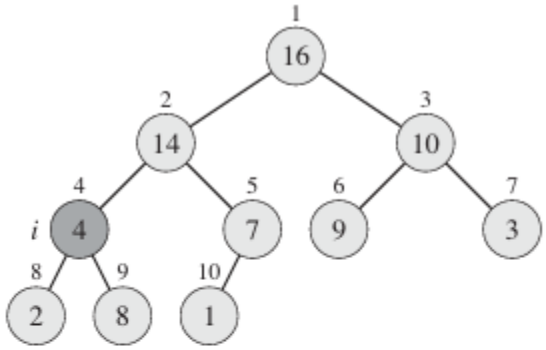
- 힙의 특징 유지
  - Pseudocode: `heapify(A,i)`

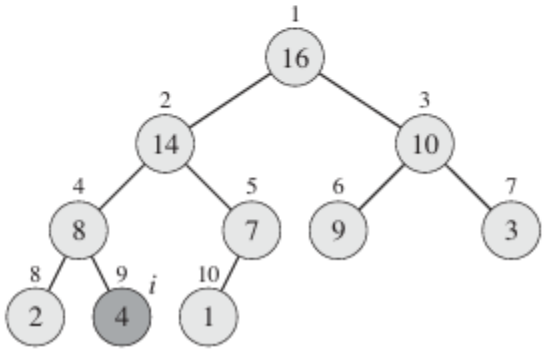
```
 1:  l = left(i)
 2:  r = right(i)
 3:  if l ≤ A.heap_size && A.heap[l] > A.heap[i]
 4:     largest = l
 5:  else largest = I
 6:  if r ≤ A.heap_size && A.heap[r] > A.heap[largest]
 7:     largest = r
 8:  if largest ≠ I
 9:     A.heap[i]와 A.heap[largest] 교환
10:     heapify(A,largest)
```

자료구조 – 힙(heap)

- 힙의 특징 유지
  - 예







Comen T.H., Leiserson C.E., Rivest R.L., Stein C., Introduction to Algorithms 3rd edition, The MIT Press, 2009.

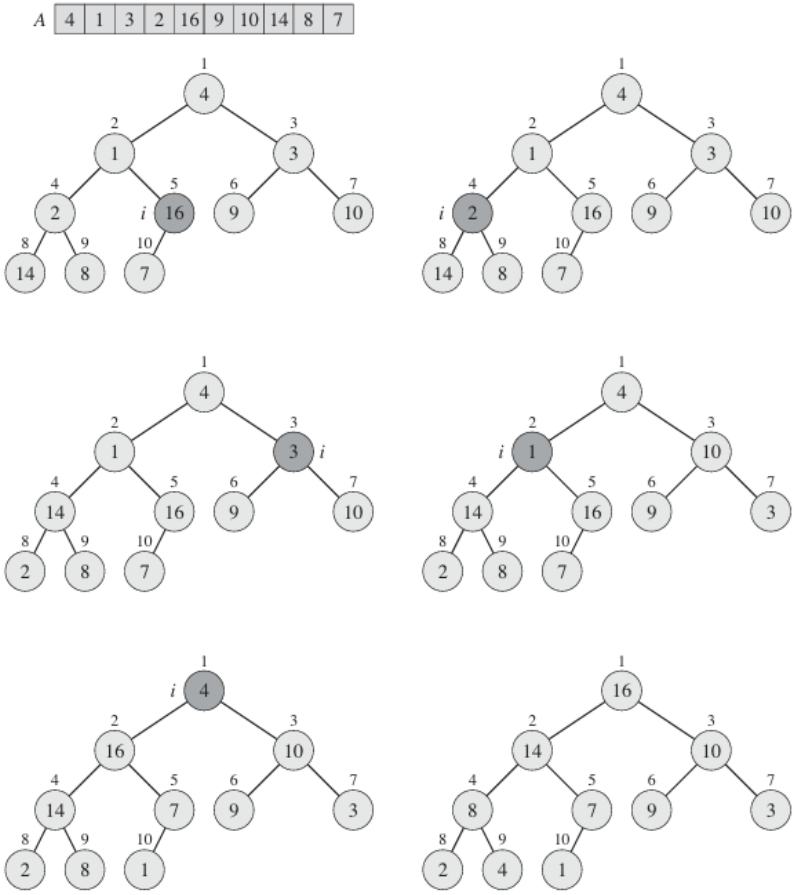
자료구조 – 힙(heap)

- 힙 만들기
  - **입력:** 임의의 배열 `arr`
  - **출력:** 최대 힙 `A`
  - `heapify()` 함수를 활용하여 구현할 수 있음
  - Pseudocode: `build_heap()`

```
1:  A.heap_size = arr의 요소 개수
2:  arr의 요소들 A.heap에 옮김
3:  for i = floor()-1 downto 0
4:      heapify(A,i)
```

자료구조 – 힙(heap)

- 힙 만들기
  - 예



Comen T.H., Leiserson C.E., Rivest R.L., Stein C., Introduction to Algorithms 3rd edition, The MIT Press, 2009.

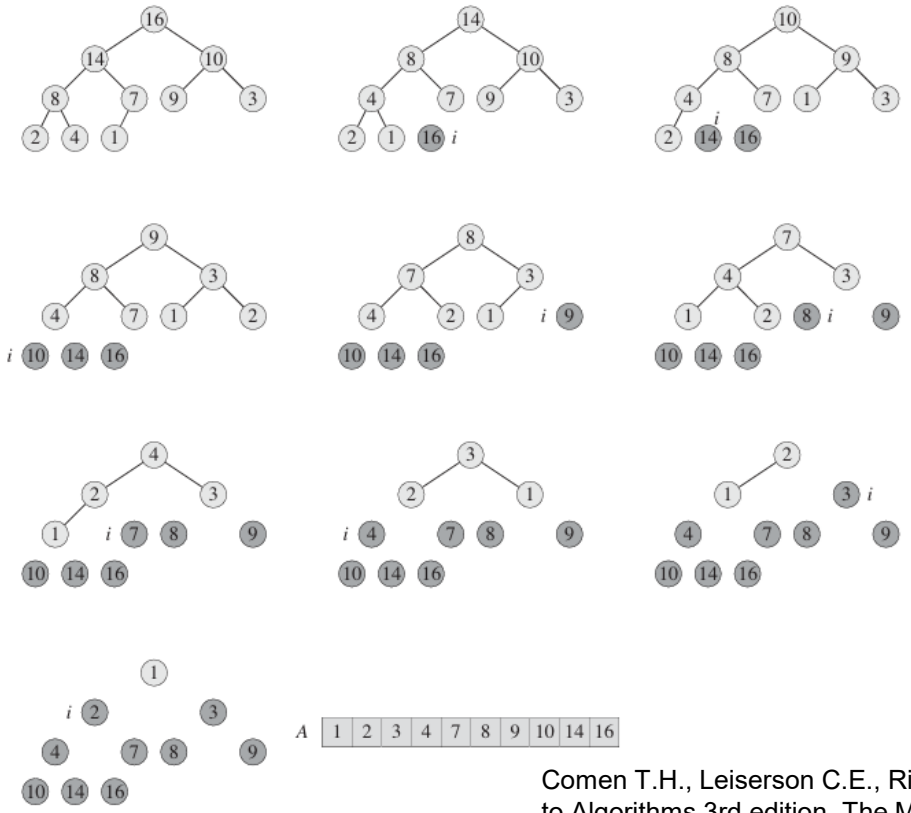
자료구조 – 힙(heap)

- 힙 정렬 알고리즘(heapsort)
  - `build_heap()`와 `heapify()` 함수를 활용하여 구현 가능
  - 배열 `arr`을 오름차순으로 정렬하는 방법
    - `heapsort(arr)`

```
1: build_heap(arr)
2: for i = arr의 요소 개수 downto 1
3:     A.heap[0]와 A.heap[i] 교환
4:     A.heap_size = A.heap_size - 1
5:     heapify(A,0)
```

자료구조 – 힙(heap)

- 힙 정렬 알고리즘(heapsort)
  - 예



Comen T.H., Leiserson C.E., Rivest R.L., Stein C., Introduction to Algorithms 3rd edition, The MIT Press, 2009.

자료구조 – 힙(heap)

- 힙에서 최댓값을 제거하여 돌려보냄
  - **입력:** 힙 `A`
  - **출력:** 힙 `A`의 최댓값
  - Pseudocode: `extract(A)`

```
1:  if A.heap_size < 1
2:      error "underflow"
3:  max = A.heap[0]
4:  A.heap[0] = A.heap[A.heap_size-1]
5:  A.heap_size = A.heap_size - 1
6:  heapify(A,0)
7:  return max
```

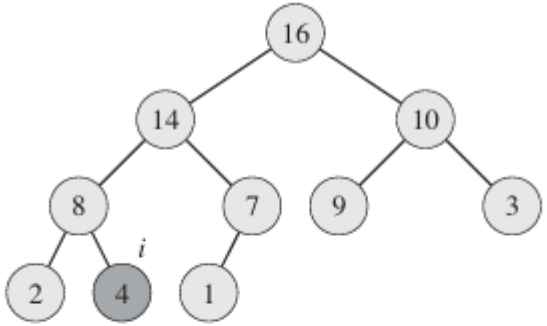
자료구조 – 힙(heap)

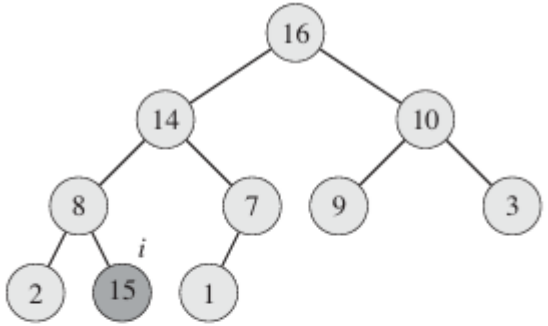
- 노드의 키 값을 증가시킴
  - **입력:** 힙 `A`, 인덱스 `i`, `key` 값(이 값까지 증가시키고 싶음)
  - **출력:** 인덱스 `i`에 기 값이 수정된 힙 `A`
  - Pseudocode: `increase_key(A,i,key)`

```
1:  if key < A.heap[i]
2:      error "새 키는 기존 키보다 작음"
3:  A.heap[i] = key
4:  while i > 0 && A.heap[parent(i)] < A.heap[i]
5:      A.heap[i]와 A.heap[parent(i)] 교환
6:      i = parent(i)
```

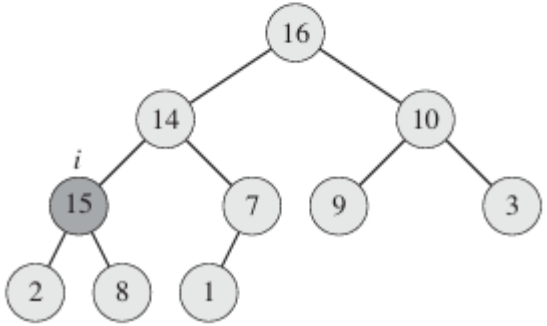
자료구조 – 힙(heap)

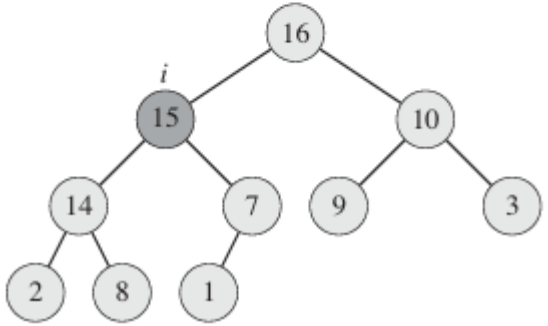
- 노드의 키 값을 증가시킴
  - 예





Comen T.H., Leiserson C.E., Rivest R.L., Stein C., Introduction to Algorithms 3rd edition, The MIT Press, 2009.





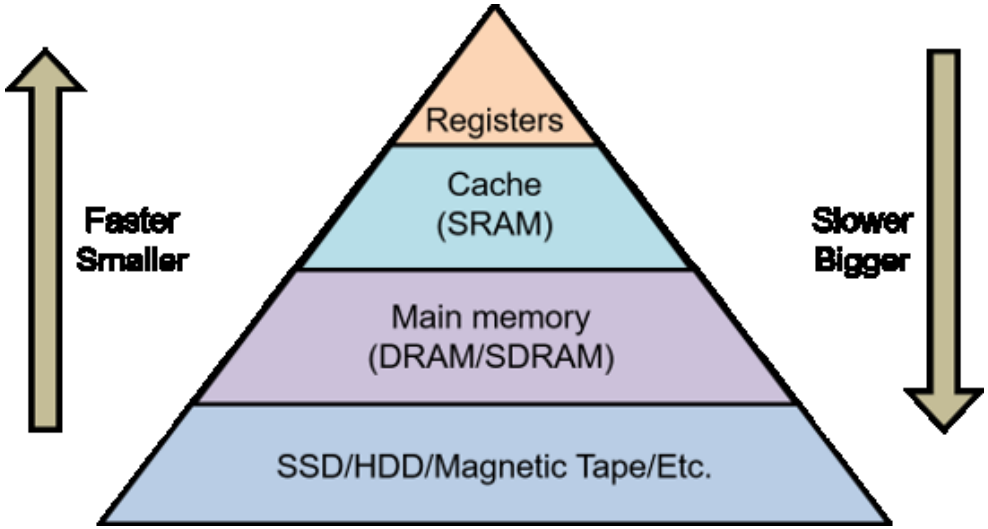
자료구조 – 힙(heap)

- 힙에 데이터를 추가함
  - **입력:** 힙 `A`, 키 `key`
  - **출력:** `key`를 추가한 힙 `A`
  - Pseudocode: `insert(A,key)`

```
1:  A.heap_size = A.heap_size + 1
2:  A[A.heap_size - 1] = 아주 작은 값
3:  increase_key(A,A.heap_size,key)
```

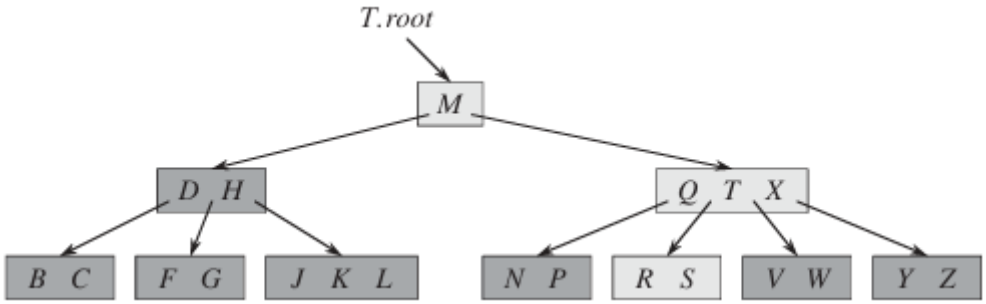
자료구조 – B-Trees

- B-trees는 balanced search trees를 의미함
  - 메모리 계층의 가장 낮은 하드디스크를 접속하는 시간이 제일 느림
    → 하드디스크 접속 횟수를 최소화해 야 함
  - 하드디스크 접속 횟수를 최소화하기 위해 만들어진 자료구조
  - 데이터베이스(SQL, NoSQL), 파일 시스템(NTFS, ext4) 등에 많이 쓰임



자료구조 – B-Trees

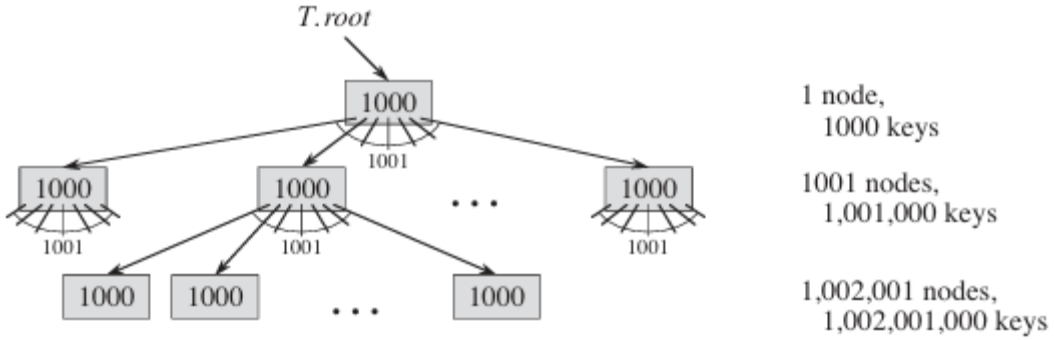
- 일반화된 이진 트리라고 부를 수 있음
  - 노드 x가 x.n 개 키를 갖고 있으며, x의 자식 노드 개수가 x.n+1
  - 노드 x에 있는 키들은 구분점 역할을 하여, x가 처리하는 키의 범위를 x.n+1 개의 하위 구간으로 나누며, 각 구간은 x의 한 자식 노드가 담당함
  - 예,
    - root 노드는 키 하나를 갖고 있어서 root의 자식이 2개가 있음
    - 왼쪽 자식 노드는 키 2개를 갖고 있어서 이 노드는 자식이 3개가 있음
      - B C **D** F G **H** J K L



Comen T.H., Leiserson C.E., Rivest R.L., Stein C., Introduction to Algorithms 3rd edition, The MIT Press, 2009.

자료구조 – B-Trees

- B-Trees를 통해 하드디스크 데이터를 불려오는 방법
  - Root 노드는 메인 메모리(RAM)에서 있음
  - 자식 노드는 하드디스크에서 있으며, 필요 시 root 노드를 통해 RAM에서 불려오게 됨
  - 예
    - 각 노드는 1000개의 키를 갖고 있으며, 1001개의 자식이 있음
    - B-Trees는 총 1,002,001,000개의 키를 저장할 수 있음
    - 어떤 키든 최대 2번의 하드디스크 접속만으로 찾을 수 있음



Comen T.H., Leiserson C.E., Rivest R.L., Stein C., Introduction to Algorithms 3rd edition, The MIT Press, 2009.

자료구조 – B-Trees

- B-Trees의 대표적인 연산
  - 탐색
  - 빈 트리 만들기
  - 트리에 키 추가
  - 트리 노드 분할
  - 트리에서 키 삭제