자료구조

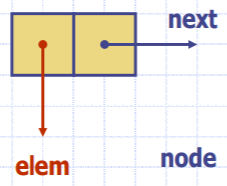
**Array list**

element를 array를 이용해서 저장할 경우 특정 위치의 값을 찾기에 편리하다. 하지만 배열의 크기가 n으로 정해질 때, n개를 넘는 요소가 들어올 경우 배열의 크기를 다시 정하는 것에서 어려움이 있다. 또한 배열의 중간에 삽입 또는 삭제를 수행할 경우 해당 위치 이후 값들에 대한 shift연산을 해야 하는 번거로움이 있다.

**Linked list** (array list의 단점들에 대한 대안)

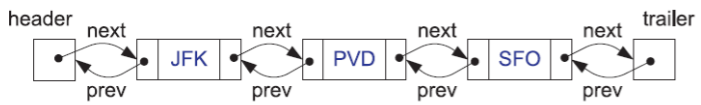
Linked list는 노드들이 선형으로 연결된 형태를 띄는 자료구조이다. element간의 연결(link)를 이용해서 리스트를 구현하기 때문에 메모리에 연속적으로 요소들을 저장할 필요가 없다. Array와 비교 시, 추가 및 삭제가 쉽다는 장점이 있지만 특정 데이터에 접근 시 최악의 경우 O(n) time 소요된다. 연결 방식에 따라 single과 double로 나뉜다.

Linked lists의 처음과 마지막 노드들은 리스트의 head와 tail이라고 부른다.

Single linked list: 한방향으로 연결된 리스트로, 각 node는 element와 다음 node를 가리키는 pointer로 구성된다.

* Insert at the head (addFront()): 1. 새로운 노드 할당 2. 새로운 노드에 element 저장 3. 새로운 노드의 pointer = old head 4. Head = new node
* Remove at the head (removeFront()): 1. Head = old head의 다음 노드 2. Old head 할당 해제
* Insert at the tail (addBack()): 1. 새로운 노드 할당 2. 새로운 노드에 element 저장 3. 새로운 노드의 pointer = null (다음 노드 없으므로) 4. Old last node의 pointer = new node 5. Tail = new node
* Remove at the tail (removeBack()): single linked list에서는 상수시간에 tail을 업데이트할 방법이 존재하지 않는다. (불충분! – double linked list에서는 상수시간에 가능하다)

Double Linked list: 노드들이 양방향으로 연결된 리스트로, 각 node는 element, 다음 node의 pointer그리고 이전 node의 pointer로 구성된다.



노드의 탐색이 양방향으로 가능하기 때문에 특정 인덱스를 탐색 시 처음과 끝 중 더 가까운 쪽부터 시작하면 되므로 탐색의 양이 single에 비해 절반 감소한다.

리스트에서 first와 last node에 더 빠른 접근을 위해 header node, trailer node를 사용하며 이들은 각각 리스트의 처음과 마지막 노드를 가리키고 어느 element를 가지지 않는다. 이러한 노드를 sentinel node or dummy node라고 부른다.

* 새로운 노드의 삽입, 삭제: single linked list와 방식 유사하며 prev pointer가 추가된 것에 유의하면 된다.

Circular Linked List: single linked list에서 마지막 노드가 리스트의 첫 번째 노드를 가리키게 하여 리스트이 구조를 원형으로 만든 자료구조. 마지막 노드를 참조하는 cursor가 single에서의 head역할을 한다.

**Repitition – loop, recursion(재귀), linear recursion, binary recursion**

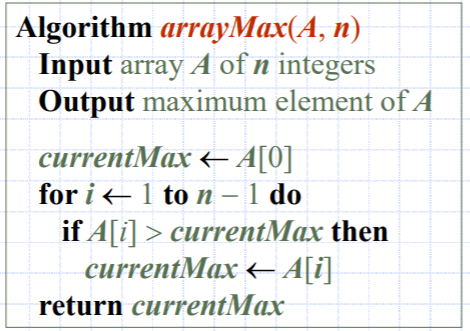
* Linear recursion: 재귀함수가 재귀적으로 호출되는 동안 각 재귀함수 내에서 호출이 한 번만 발생하는 재귀 (ex. Sum)
* Tail Recursion: linear recursion의 한 형태로 tail recursion은 마지막 함수 호출에서 반환하는 값이 그 재귀 호출의 반환 값과 동일 경우를 말함. 즉 반환 값이 병합(+,/…)되지 않고 마지막 반환 값이 계속 유지되어 반환 된다.
* Binary recursion: 재귀 함수내에서 함수 호출이 두개 이상인 재귀 (ex. 피보나치 수열)

**Algorithm 분석**

Running time: 대부분 알고리즘의 input size에 비례한다. 평균 수행 시간을 측정하기 어렵기 때문에 worst case 수행시간에 초점을 맞춘다.

실험을 통한 연구: 알고리즘을 실제로 구현하여 input의 크기나 값을 변경하면서 결과를 측정. 두개의 다른 알고리즘을 비교하기 위해서는 하드웨어와 소프트웨어 환경이 같아야 한다.

이론적 분석(Theoretical): 구현이 아닌 pseudocode와 같은 high-level description을 사용하며 input size를 n으로 고정한다. 하드웨어와 소프트웨어 환경에 독립적이다.

Pseudocode: 높은 수준의 description으로 프로그램 디자인 이슈를 숨긴다.

이와 같은 형태 🡪

Primitive Operations: 알고리즘에 의해 수행되는 기본 computation들(변수에 값 할당, 배열에 indexing, 함수 호출, 함수 반환 등)로 pseudocode에서 쉽게 식별 가능하며 프로그래밍 언어에서 크게 독립적이다. 상수시간이 걸리며 최대 수행 연산의 개수는 input size의 함수로 표현됨

Running time T(n)은 primitive operation의 최대 수행 연산의 개수(worst case)와 수행 시간(생략)의 계산으로 구할 수 있다. T(n)은 상수 factor와 growth rate가 있으며 constant factor 경우 하드웨어/소프트웨어 환경이 변하면 바뀔 수 있지만 growth rate는 어느 경우라도 동일하다. 따라서 running time은 T(n)의 growth rate에 대해 나타낸다. 🡺 big-Oh 사용하여 표기

Big-Oh: O(g)는 G보다 기울기가 더 낮거나 같은 함수들의 집합을 의미. Ex) O(n) = {n, 2n+1, ..}

Big-Omega: Ω(g)는 g보다 기울기가 더 높거나 같은 함수들의 집합을 의미

Big-Theta: Θ(*g*)는 g와 기울기가 같은 함수들의 집합을 의미(big Oh와 big Omega의 교집합)

ADT (Abstract Data Types): 자료 구조의 추상화로 저장된 data, data에 대한 연산, 연산과 관련된 에러 조건들을 명시한다.

**Stack -** LIFO원칙에 따라 자료를 삽입 또는 삭제하는 자료구조

Stack operation: push(object), pop(), top(), size(), empty()

Exceptions: ADT의 연산 수행 시 에러를 일으키는 경우 exception을 호출한다. stack경우 pop(), top()함수에서 stack이 빈 상태일 경우 수행될 수 없으므로 StackEmpty exception을 호출한다.

C++ Run-Time Stack: c++ run-time system은 stack을 이용하여 active function들의 chain의 track을 유지한다. 함수가 호출될 때, 시스템은 stack에 지역변수, 반환 값, pc, 함수의 현 실행 지점 정보를 포함하는 하나의 frame을 push한다. 함수가 종료되면 해당 frame은 pop되고 stack의 top함수가 수행된다.

Array-based Stack

- stack을 array를 이용하여 구현하는 것으로 top 요소의 index 값을 변수에 저장해 둔다. (t: top 요소의 index, -1로 초기화)

- Size(): return t+1 | empty(): return (t<0) | top(): return S[t] | push(e): t🡨t+1, S[t]🡨e | pop(): t🡨t-1

- StackEmpty exception: top(), pop() | StackFull exception: push(e)

- 성능: stack의 요소 개수 n일 때, 공간복잡도: O(n), 각 연산에 대한 시간복잡도: O(1)

- 한계: stack의 최대 크기가 처음에 정해져야 하며 바꿀 수 없다. 이러한 이유로 full stack에 push를 수행할 경우 implementation-specific exception을 불러일으킨다. (배열 사용에 의해 발생 문제)

Linked List-based Stack

- Stack을 single linked list로 구현하며 현재 stack의 크기를 변수 n에 저장한다. (초기값 0, S: single linked list class), 배열과 달리 full stack문제가 발생하지 않는다.

- Size(): return n | empty(): return n==0 | top(): S.front() | push(e): S.addFront(), n🡨n+1 | pop(): n🡨n-1, S.removeFront()

- StackEmpty exception: top(), pop()

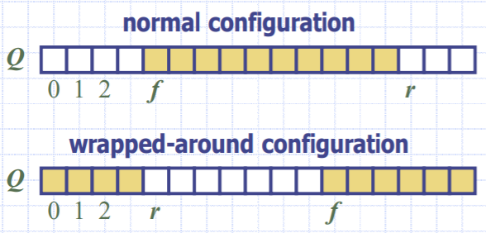
- 성능: 공간복잡도: O(n), 각 연산에 대한 시간복잡도: O(1)

**Queue -** FIFO원칙에 따라 자료를 삽입 또는 삭제하는 자료구조

Queue operations: enqueue(object) (queue의 뒤에 삽입), dequeue() (queue의 앞부분 삭제), front() (제거 없이 맨 앞 요소 값 반환), size(), empty()

Exceptions: dequeue()와 front()연산에서 큐가 빈상태인 경우 QueueEmpty exception이 발생한다.

Array-based Queue

- queue를 array를 이용하여 구현한 것으로 배열 사용 시 크기가 N인 circular array를 사용한다. (f: 맨 앞 요소의 index, r: 맨 뒤 요소 바로 다음 index, n: queue의 요소 개수(크기와 빈 여부 결정하기 위해 사용))

- size(): return n | empty(): return n==0 | front(): Q[f] | dequeue(): f🡨(f+1) mod N, n=n-1 | enqueue(e): Q[r]🡨e, r🡨(r+1) mod N, n=n+1

- QueueEmpty exception: front(), dequeue() | QueueFull exception: enqueue(e)

- QueueFull exception 경우, 구현 시 array를 사용할 때 발생하는 문제이다. (implementation-dependent)

**Vector (Array list)** – index를 이용하여 element에 대한 접근, 삽입, 삭제를 지원하는 sequence이다.

Vector method: at(i) (index i에 있는 요소 반환), set(i, o) (index i에 있는 요소를 o로 교체), insert(i, o) (index i를 가지는 새로운 요소 o 삽입), erase(i) (index i 요소를 삭제), size(), empty()

Exceptions: 부정확한 index가 주어질 경우 예외처리

Array-based Implementation

- 크기가 N인 배열 A를 사용. (n: 저장된 요소의 개수)

- at(i): return A[i] | set(i, o): A[i] = o | insert(i, o): index i~n-1의 값을 한칸씩 뒤로 이동, A[i]🡨o, n🡨n+1 | erase(i): index i+1~n-1 값을 한칸씩 앞으로 이동, n🡨n-1

- 성능

insert, erase 연산을 제외한 나머지 연산의 수행시간 O(1) time | insert, erase연산의 경우 worst case(i=0)에 대해 수행시간 O(n) time | 공간복잡도 O(N)

circular array를 사용할 경우 i=0일 때 insert, erase연산의 수행시간 O(1) time 가능

insert연산 경우, 배열이 full이면 예외처리를 하는 대신 더 큰 배열로 배열을 교체한다. (c++ 구현 시, 이 경우에 용량 2\*N인 새로운 배열 B를 할당하여 기존 배열A의 값들을 B로 옮긴 뒤, A의 할당을 해제 후 배열 B를 가리키는 A를 새로 할당한다.)

STL vector – vector header에 존재

연산: size(), empty(), resize(n), reserve(n), operator [i], at(i) (V[i]와 달리 범위를 넘을 경우 예외처리), front(), back(), push\_back(e), pop\_back()

**Lists –** index대신 node를 사용하여 element에 대한 접근과 삭제를 지원하는 sequence

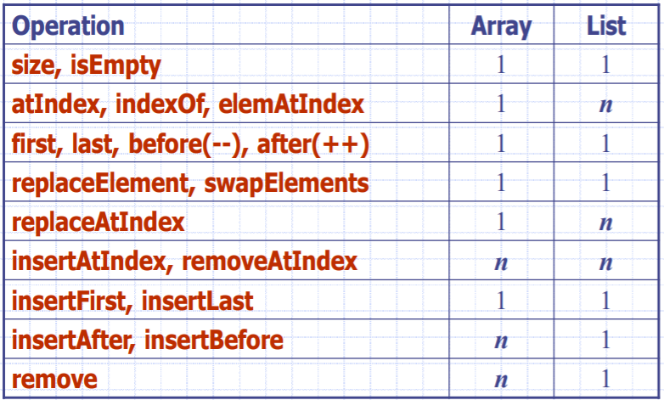
List method: size(), empty(), begin(), end(), insertFront(e) (= insert(L.begin(), e)), insertBack(e) (=insert(L.end(), e)), removeFront(), removeBack(), insert(p, e) (position p 전에 요소 e를 삽입), remove(p) (= erase(p))

Single linked list 경우 node는 element, next node의 pointer로 구성된다.

Double linked list 경우 node는 element, next node의 pointer, prev node의 pointer로 구성된다.

Double linked list 성능: 공간복잡도 – O(n) (n:요소의 개수) , 모든 연산의 시간 복잡도 – O(1)

**Sequence ADT** – Vector와 List ADT의 합집합으로 요소들이 index 나 position(node)로 접근된다.

Method: size(), isEmpty(), 벡터 기반 { elemAtIndex(i), replaceAtIndex(i, o), insertAtIndex(i, o), removeAtIndex(i), first }, list 기반 { first(), last(), before(p), after(p), replaceElement(p, o), swapElements(p, q), insertBefore(p, o), insertAfter(p, o), insertFirst(o), insertLast(o), remove(p) }, atIndex(i), indexOf(p)

Sequence의 활용: stack, queue vector or list에 대한 광범위 교체, 작은 db, 더 복잡한 자료구조를 세울 때 사용된다.

Array-based: 각 node를 저장하기 위해 순환 배열을 사용한다. 순환 배열에 node들 저장, 각 node는 요소와 index 저장

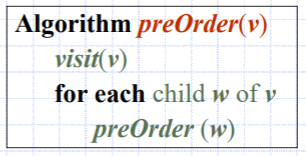
**Tree**

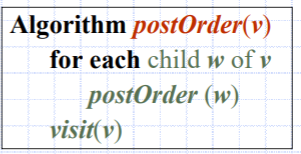
List, vector와 같은 선형 구조와 달리 tree는 계층적 구조를 가진다. 계층적 구조이기 때문에 노드간 부모-자식 관계가 존재한다.

용어

* Root: tree에서 부모가 없는 최상위 노드
* Internal node: 1개 이상의 자식 node가 존재하는 node
* External node: 자식이 존재하지 않는 node로 leaf를 뜻한다.
* Ancestors of a node: 한 노드의 부모를 포함한 조상 노드들
* Descendant of a node: 한 노드의 바로 연결된 자식과 그 밑 자식들
* Depth: 한 노드의 조상의 수
* Height: 노드들의 depth중 가장 큰 depth로, tree의 높이다.
* Subtree: tree의 한 노드와 그 후손들로 구성된 하나의 트리 (트리안에 트리)
* Sibling: 같은 부모노드를 가진 자식 노드들의 관계

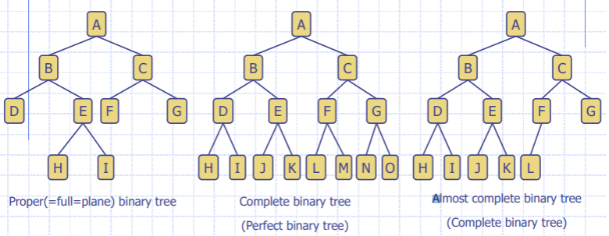
Tree ADT: size(), empty(), root() (root의 position), positions() (모든 노드들의 위치 list), parent(), children(), isRoot(), isExternal()

트리의 순회 방법 (Traversal)

* Preorder Traversal: 부모 -> 자식 순으로 방문한다. 주로 책의 목차를 print할 때 사용.
* Postorder Traversal: 자식들 먼저 방문 -> 부모 방문으로 진행한다. 주로 하위 디렉토리들이 차지하는 공간의 용량을 계산할 때 사용.

Binary Tree: 모든 노드가 최대 두개의 자식 노드를 가진 트리를 의미한다. 자식을 left child, right child라고 칭함.

Binary tree 종류

* Proper(=full=plane) binary tree: 모든 노드가 0 or 2개의 자식을 가진 tree
* Complete(=perfect) binary tree: 모든 internal 노드가 2개의 자식을 가지며 leaf 노드들의 깊이가 모두 같은 tree)
* Almost complete(=complete) binary tree: complete tree에서 leaf노드들의 오른쪽부터 순차적으로 지워져 있는 tree
* Complete binary tree를 perfect binary tree라 지칭할 경우 almost complete를 complete라고도 부름

Binary tree의 응용: 수식 표현, decision tree, searching(binary search tree)

BinaryTree ADT: Tree ADT의 함수들에 추가로 left(), right()함수 존재.

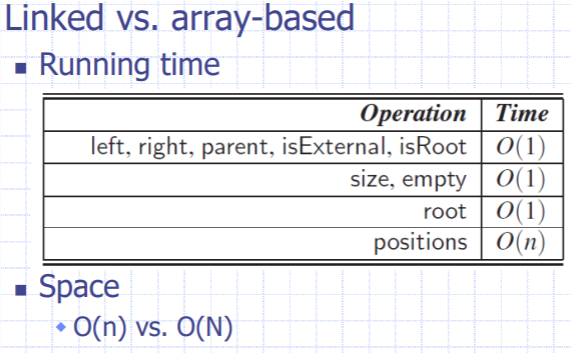
Binary Tree의 Traversal

* Preorder: 부모 -> 왼쪽 자식 -> 오른쪽 자식 순서로 순회
* Postorder: 왼쪽 자식 -> 오른쪽 자식 -> 부모 순서로 순회
* Inorder: 왼쪽 자식 -> 부모 -> 오른쪽 자식 순서로 순회. 이진 트리 그림 그릴 때 사용
* 수식 표현 트리에서 inorder를 사용하면 사람이 보기 쉽게 수식 나열, postorder로 계산
* Euler Tour Traversal: 주어진 트리에 대해 모든 간선을 정확히 한번씩만 방문하도록 하는 순회이다. 시작점과 끝점이 root로 동일하다. 각 노드당 3번 방문 되므로(왼쪽 자식 방문 전, 왼쪽 자식 방문 후 오른쪽 자식 방문 전, 오른쪽 자식 방문 후) 한 노드당 하는 일의 수가 모드 3개로 일정하여 이 순회의 시간 복잡도는 O(n)이다. 일반적인 종류의 알고리즘을 쉽게 표현할 수 있는 장점이 있다.

Linked Structure for Binary Trees: 각 노드는 element, parent node, left child, right child로 구성

Array-based Binary Tree: 노드들이 깊이 순서로 배열에 저장되며 root=1, 한 노드의 인덱스가 i일때, 왼쪽 자식의 index = 2\*I, 오른쪽 자식의 index = 2\*i+1로 각 노드에 접근할 수 있다.

성능:



**Priority Queue** – 우선순위에 의해 데이터가 정렬되기 대문에 queue에 삽입되는 요소의 순서가 중요하지 않고 데이터 속성에 따라 제거가 가능하다.

Priority Queue ADT: entry는 (key, value) 쌍으로 구성된다. Key는 우선순위를 가리킴

* Method: insert(e), removeMin() (= extractMin()), min(), size(), empty()

STL priority queue class: #include<queue>

* size(), empty(), push(e), top(), pop()

List를 이용한 우선순위 큐 구현: list를 정렬된 상태로 저장, 정렬되지 않은 상태로 저장 비교

|  |  |  |
| --- | --- | --- |
| Operation | Unsorted List | Sorted List |
| Size, empty | O(1) | O(1) |
| Insert | O(1) | O(n) |
| Min, removeMin | O(n) | O(1) |

우선순위 큐를 이용하여 크기가 n인 배열을 정렬: Selection-Sort, Insertion-Sort

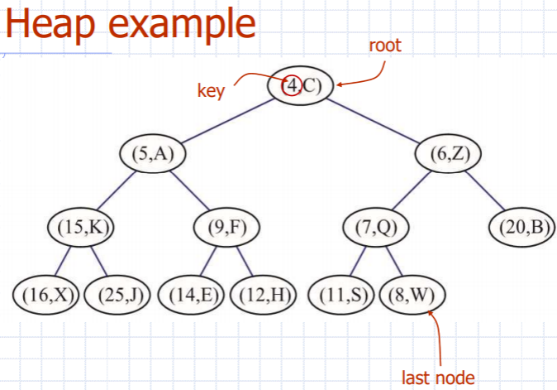
1. 배열의 값을 모두 우선순위 큐에 넣기 (insert) – n번 반복
2. 우선순위 큐에서 값을 하나씩 기존 배열로 옮기기 (removeMin) – n번 반복

정렬을 위해 insert, removeMin연산을 이용하며 연산의 구현 방식에 따라 정렬의 성능이 다르다.

|  |  |  |
| --- | --- | --- |
|  | 구현 | 시간 복잡도 |
| Selection-Sort | Unsorted List를 이용한 PQ | O(n^2) |
| Insertion-Sort | Sorted List를 이용한 PQ | O(n^2) |

In-place Insertion-Sort: PQ 사용 안하고 배열 앞부터 정렬된 부분과 비교해 요소 위치 찾아 삽입

**Heap –** 최댓값, 최솟값을 찾아내는 연산을 쉽게 하기 위해 고안된 자료형

각 node가 (key, element)로 구성되며 Heap-Order와 (Almost) Complete Binary Tree조건을 만족하는 binary tree이다.

\* heap-order: 모든 노드에 대해 각 노드의 키값이 자식 노드의 키값보다 작지 않거나(max heap, 이때 root의 키값이 가장 크다) 자식노드의 키값보다 크지않은(min heap, root의 키값이 가장 작다)을 만족해야 한다. (우리는 min heap)

\* last node: heap의 maximum 깊이의 가장 오른쪽 노드를 의미

삽입 연산

1. 완전 이진 트리 구조를 유지하기 위해 삽입 시 트리의 가장 마지막에 원소를 추가한다.
2. 그 후 삽입된 형태의 힙이 힙의 조건(heap order)를 만족하도록 재복구시킨다.

복구 방법: Upheap (새로운 key k 삽입 후, heap-order 조건 위반한 경우)

* 삽입 노드로부터 root 방향으로 연결된 노드들을 따라 k를 각 노드와 swap하는 방식
* Key k가 root에 도달하거나 부모노드의 key값이 k와 같거나 작은 경우 upheap을 종료
* Heap의 높이가 O(log n)이므로, upheap의 수행시간은 O(log n) time이다.

삭제 연산 (min heap 경우, 최소 키값을 가진 node를 삭제해야하므로 root를 삭제한다)

1. Root key를 힙의 last node의 값으로 교체하고 last node를 제거한다.
2. 그 후 heap-order 조건을 만족하도록 재복구시킨다.

복구 방법: Downheap (root값을 last node의 키 k로 교체 후, heap-order 조건 위반한 경우)

* Root부터 더 작은 값을 가진 자식 노드방향으로 k를 각 노드와 swap하는 방식
* Key k가 leaf에 도달하거나 자식노드의 key값이 k와 같거나 더 클 경우 downheap 종료
* 수행시간: O(log n) time

Heap-Sort (heap을 이용하여 우선순위 큐 구현)

Heap을 이용한 우선순위 큐: size(), empty() – O(1) time | insert, removeMin – O(log n) time

heap기반에 우선순위큐를 이용하여 크기가 n인 배열을 정렬시 O(n log n) time 걸린다. 즉, list 기반에 우선순위큐 방식보다 정렬 속도가 빠르다. | 실제 우선순위 큐는 heap기반으로 구현됨

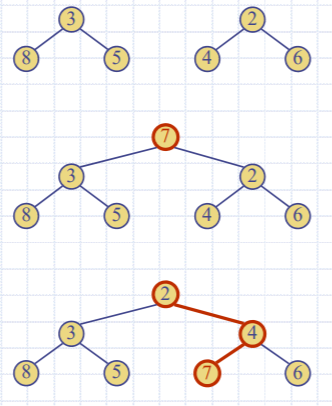
Vector-based heap 구현

* 배열을 이용하여 heap을 구현하는 방식으로, 배열에 각 노드 저장 방식은 binary tree와 동일
* 삽입 연산시 n+1 위치에 값 삽입, 최소값 제거 연산시 n위치 삭제

Heap 구현 방식

Top-Down heap 구현: 빈 힙에 위에서부터 아래로 노드를 삽입하면서 확장하는 방식으로 insert연산을 n번 수행 (n: 노드의 개수) | O(n log n) time

Bottom-up heap 구현: O(n) time

* n개의 노드가 미리 주어질 때, 힙을 더 빠른 속도로 구현할 수 있는 방법이다.
* 들어오는 키 값 순서대로 트리를 구성한 뒤, 힙 구조가 되도록 마지막 높이에서 한단계씩 높이를 올라가며 heap-order를 만족하는지 검사하여 재조정한다. downheap이용
* 두개의 힙을 합쳐 새로운 힙을 완성해나간다. 만약 두개의 힙과 키 k가 주어질 때, k를 root node로 하고 subtree로 주어진 두개의 힙을 가지는 새로운 힙을 생성한 뒤 heap-order조건을 만족하도록 downheap을 수행해 힙을 재조정한다. 이 과정을 계속 반복하며 n개의 키를 가진 heap을 완성한다.

**Map –** entry가 (key, value)쌍으로 이루어진 자료구조로 데이터 검색시 유용하다. (key는 unique)

Map ADT: find(k) (key k를 가진 entry를 찾아 value를 반환), put(k, v) (키값이 k인 entry존재시 value를 v로 변경, 없을 경우 삽입), erase(k), size(), empty()

Map 구현시 두가지 방법이 존재한다 : list기반, array기반

* list기반 map 성능: put, find, erase 연산 수행시 worst case경우 O(n) time 걸림(값 찾을 때 선형으로 찾으므로) , map구현시 좋지 않다
* array기반 map은 hash-table을 사용한다.

**Hash Table :** 연관배열 구조(키와 값이 1:1로 연관되어 있는 구조)를 이용하여 키에 대한 결과 값을 저장하는 자료구조

Hash table은 hash function과 크기 N인 bucket array(or table)로 구성된다.

Hash funciton: 키를 랜덤한 방법으로 가능한 충돌이 발생하지 않도록 분산하는 것을 목표로 한다. 다양한 길이의 키 k를 [0,N-1]사이 정수값을 가지는 일정한 길이의 값 h(k)로 변경하여 저장소를 효율적으로 운영할 수 있도록 한다. H(k)는 k의 해쉬값이며, bucket array의 index로 사용된다.

* Hash code: key를 integer로 형변환 | h1(k)
* Compression Function: 정수형인 h1(k)를 [0, N-1]범위 정수값으로 매핑하는 함수 (N:소수값 – 해쉬값의 분배를 randomize하기 위해) | Division과 MAD 방식이 존재

Division: h2(h1(k)) = |h1(k)| mod N

MAD(Multiply, Add and Divide): h2(h1(k)) = (a\*h1(k)+b) mod N (a, b >= 0, a mod N != 0), divison보다 더 정교한 방법으로 키값간 충돌이 덜 발생한다.

* Collision Handling: 해쉬값을 테이블의 index로 사용하기 때문에 서로 다른 키 값이 같은 해쉬값을 가질 때 충돌이 생긴다. 충돌을 해결하기 위한 방법은 아래와 같다.

Separate Chainig: table의 각 cell을 linked list로 구현하여 entry를 저장 | 구현이 간단하지만 테이블이외에 추가 메모리를 사용

Open addressing: 충돌되는 item을 테이블의 다른 cell에 저장하는 방법 | linear probing, double hashing

Linear probing: 충돌되는 item을 한칸씩 이동하며 다음 이용가능한 cell에 저장 | primary clustering(충돌 발생시, 뒤 슬롯에 데이터를 넣어 하나의 데이터 덩어리를 이루기 때문에, 데이터들이 특정 위치에만 밀집하는 현상을 말한다. 때문에 slot이 많아질수록 탐색 시간이 엄청 늘어난다) 문제점이 존재.

Double hashing: linear probing에 의해 발생되는 clustering문제를 해결한 방법으로 보조 해시 함수 d(k)를 이용한다. 테이블의 index = (h(k) + j\*d(k)) mod N (j=0,1,…,N-1)이며, 충돌이 발생할때마다 j의 값을 0부터 하나씩 증가하며 index를 새로 구한다.

성능: worst case경우(모든 key에 대해 충돌발생), n개의 entry를 가진 hash table에서 map 연산은 O(n) time 걸리지만, 보통은 O(1) time이 소요된다.

Dictionary: entry가 (key, element)로 구성되며 중복된 key를 허용한다.

**Binary Search Tree**

Search Tree: ordered maps과 ordered dictionaries를 구현하는데 사용되는 tree 자료구조

ADT method: find(k), put(k, v), erase(k), size(), empty()

Binary Search Tree: internal node에 key or key-value entry를 저장하고 key(u)<=key(v)<=key(w) (u: v의 left node, w: v의 right node) 조건을 만족하는 이진 트리 자료구조이다. (단, external node에는 값을 저장하지 않는다)

Binary search tree의 inorder traversal은 증가하는 순서로 key를 방문한다.

삽입 연산: key k에 대한 탐색을 한다. leaf노드에 도달 시 k가 존재하지 않은 것으로 leaf노드에 k를 삽입하여 internal node로 확장한다.

삭제 연산: tree에 key k가 존재할 때, 해당 노드를 찾은 후 만약 leaf 자식 노드를 가진 경우 해당 노드와 leaf 자식 노드를 둘다 지운다. RemoveExternal(w)함수를 이용. 만약 k가 저장된 노드의 자식노드가 모두 internal 노드인 경우 1. 오른쪽 자식 중에서 가장 작은 값을 해당 위치로 옮기기 2. 왼쪽 자식 중에서 가장 큰 값을 해당 위치로 옮기기 방법이 있으며 둘중 1을 더 많이 쓴다.

성능: n개의 item을 가진 ordered map을 높이가 h인 binary search tree로 구현시 공간복잡도 O(n), 삽입, 삭제, 찾기 연산시 O(h)이 걸린다. Worst case에서 h=n , best case에서 h=log n

**AVL Tree**