자료구조

**ADT: 자료구조의 추상화**

**Array**

* 장점: 배열에서 특정 위치의 값을 찾기에 편리하다.
* 단점: 배열의 크기를 넘는 값을 삽입할 경우 문제 / 배열의 중간에 삽입 or 삭제 연산 시 번거로움 발생

**Linked List**

* 선형으로 연결된 노드들을 가지는 자료구조
* 노드간 pointer를 이용해 list를 구현하므로 메모리에 연속적으로 값을 저장할 필요 없다.
* Array list와 달리, 추가 및 삭제가 쉽지만 데이터 접근 시 최악의 경우 O(n) time 소요

1. Single Linked list

* 한방향으로 연결된 리스트로, 각 node는 element와 다음 node의 pointer로 구성
* ADT method: empty(), front(), addFront(), removeFront()

1. Double Linked list

* 양방향으로 연결된 리스트로, 각 node는 element, next pointer, prev pointer로 구성
* 노드의 탐색이 양방향으로 가능하므로 탐색 양이 single에 비해 절반 감소한다.
* ADT method: empty(), front(), back(), addFront(), removeFront(), addBack(), removeBack()

1. Circular Linked list

* Single linked list의 구조를 원형으로 만든 자료구조
* 마지막 노드를 참조하는 cursor가 single에서의 head역할을 한다.

**Algorithm 분석**

1. Running time: 대부분 알고리즘의 input size에 비례하며 평균수행시간을 측정하기 어렵기 때문에 worst case 수행시간에 초점을 맞춘다.
2. 실험적 분석방법
   1. 알고리즘 구현이 필수적, 어려운 알고리즘 경우 구현이 어렵다.
   2. 실험에 포함되지 않는 input에 대해 알고리즘 수행시간을 알 수 없다.
   3. 반드시 같은 하드웨어와 소프트웨어 환경에서 분석이 이루어져야 한다.
3. 이론적 분석방법
   1. 알고리즘 구현 대신에 pseudocode와 같은 high-level description을 사용
   2. Input size를 n으로 고정하여 n에대한 함수로 수행시간 표현
   3. 모든 가능한 input에 대해 고려할 수 있다.
   4. 하드웨어와 소프트웨어 환경에 독립적으로 알고리즘 수행시간을 평가할 수 있다.
4. 이론적 분석방법을 통한 알고리즘 수행시간 계산
   1. Primitive operation: +, -, \*, /, 비교연산과 같이 단위시간에 수행될 수 있는 연산
   2. 알고리즘에서 사용되는 primitive operation의 수를 카운트해서 input size n에 대한 함수 T(n)로 정의
   3. Growth rate는 하드웨어와 소프트웨어의 환경에 의해 변하지 않으므로 running time은 T(n)의 growth rate에 대해 나타낸다 🡺 big-Oh 사용하여 표기
5. **asymptotic notation** (점근적 표기법)
   1. Big-Oh: O(g)는 G보다 기울기가 더 낮거나 같은 함수들의 집합
   2. Big-Omega: Ω(g)는 g보다 기울기가 더 높거나 같은 함수들의 집합을 의미
   3. Big-Theta: Θ(g)는 g와 기울기가 같은 함수들의 집합을 의미(big Oh와 big Omega의 교집합)

**Stack**

* LIFO원칙에 따라 자료를 삽입 또는 삭제하는 자료구조
* ADT method: push(object), pop(), top(), size(), empty()
* Exceptions: stack이 빈 상태일 때, pop(), top()연산 수행 시 StackEmpty exception 발생
* C++ Run-Time Stack: c++ run-time system은 stack을 이용하여 active function들의 chin track을 유지한다. 함수 호출 시 시스템은 호출된 함수에 관한 정보(지역변수, 반환 값, pc, 현 실행지점)를 포함하는 하나의 frame을 stack에 push한다. 함수 종료 시 해당 frame이 pop되고 stack의 top함수가 수행됨

1. Array-based Stack

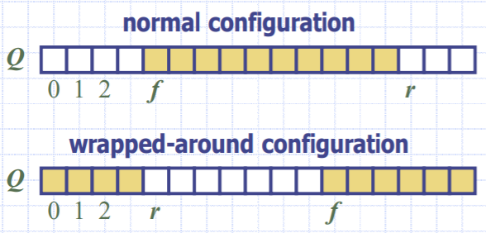
* 공간 복잡도: O(n)
* 시간 복잡도: push(object), pop(), top(), size(), empty() 모두 O(1)
* 한계: 배열을 사용하기 때문에 stack크기 조정 불가능하다. 따라서 full stack에 대해 push()연산 수행 시 StackFull exception발생 🡸 implementation-specific exception

1. Linked list-based Stack

* Single linked list로 구현
* 배열과 달리 full stack문제가 발생하지 않는다.
* 공간 복잡도: O(n)
* 시간 복잡도: 모든 연산에 대해 O(1)

**Queue**

* FIFO원칙에 따라 자료를 삽입 또는 삭제하는 자료구조
* ADT method: size(), empty(), enqueue(object), dequeue(), front()
* Exceptions: 빈 큐에 대해 dequeue()와 front()연산 수행 시 QueueEmpty exception 발생

1. Array-based Queue

* 크기가 N인 circular array를 사용한다.
* Full queue에 대해 enqueue(e)연산 수행 시 QueueFull exception발생 🡸 implementation-specific exception
* 공간 복잡도: O(N)
* 시간 복잡도: 모든 연산에 대해 O(1)

1. Linked list-based Queue

* Circular linked list이용하여 구현
* 공간 복잡도: O(n) (n: queue에 저장된 요소의 개수)
* 시간 복잡도: O(1)

**Vector**

* Index를 이용하여 element에 대한 접근, 삽입 그리고 삭제를 지원하는 자료구조
* ADT method: at(i), set(i, object), insert(I, object), erase(i), size(), empty()
* Exceptions: 부정확한 index가 주어질 경우 예외처리

1. Array-based 구현

* 크기가 N인 배열 사용하며 저장된 요소의 개수를 변수 n에 저장
* 공간 복잡도: O(N)
* 시간 복잡도

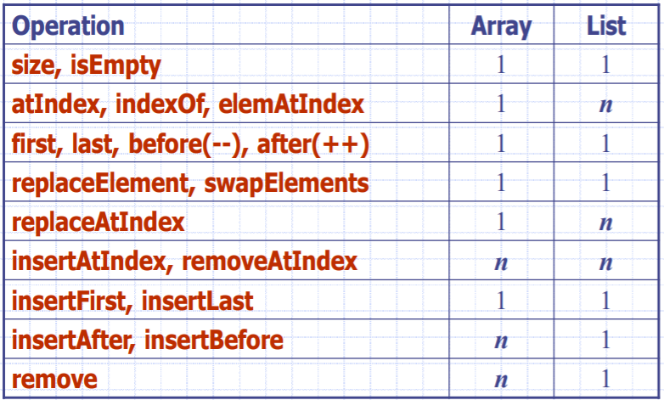
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Size() | Empty() | At(i) | Set(i, e) | Insert(i, e) | Erase(i) |
| O(1) | O(1) | O(1) | O(1) | O(n) | O(n) |

* 배열이 full일 때 insert()연산을 수행할 경우: 예외처리를 하는 대신 더 큰 배열로 배열을 교체한다.

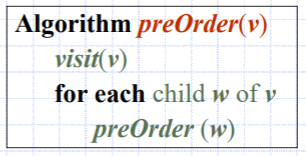
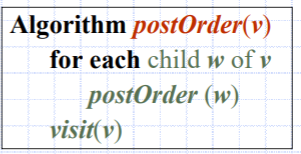
🡸 배열 크기 확장 ①상수만큼 더 큰 배열 ②기존배열의 두배

**Sequence ADT**

* Vector와 List ADT의 합집합으로 element에 index나 position(node)로 접근한다.
* ADT method (vector와 list ADT의 함수들을 모두 제공)
  1. Size(), isEmpty()
  2. 벡터 기반: elemAtIndex(i), replaceAtIndex(i, object), insertAtIndex(i, object), removeAtIndex(i)
  3. 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)
  4. Indext와 position사이에 bridging함수: atIndex(i), indexOf(p)
* Array기반 vs list기반 시간 복잡도

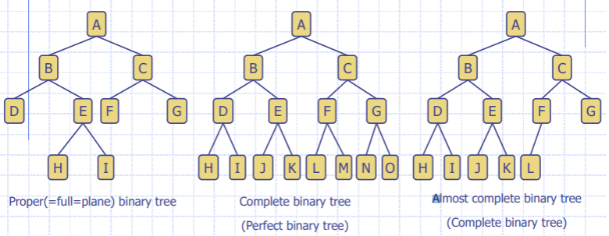


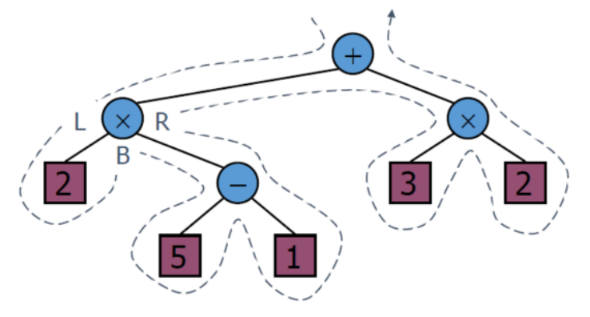
**Tree**

* List와 vector가 선형구조인 반면, tree는 계층적 구조를 가진다.
* 노드간 부모-자식 관계가 존재
* 용어
  1. Root: tree에서 부모가 없는 최상위 노드
  2. Internal node: 1개 이상의 자식 node가 존재하는 node
  3. External node: 자식이 존재하지 않는 node로 leaf를 뜻한다.
  4. Ancestors of a node: 한 노드의 부모를 포함한 조상 노드들
  5. Descendant of a node: 한 노드의 바로 연결된 자식과 그 밑 자식들
  6. Depth: 한 노드의 조상의 수
  7. Height: 노드들의 depth중 가장 큰 depth로, tree의 높이다.
  8. Subtree: tree의 한 노드와 그 후손들로 구성된 하나의 트리 (트리안에 트리)
  9. Sibling: 같은 부모 노드를 가진 자식 노드들의 관계
* ADT method: size(), empty(), root(), positions() (모든 노드들의 위치 list), parent(), children(), isRoot(), isExternal()
* 트리의 순회(Traversal) 방법
  1. Preorder Traversal: 부모 🡪 자식(왼->오)순으로 방문한다. 주로 책의 목차를 print할 때 사용
  2. Postorder Traversal: 자식 🡪 부모순으로 방문한다. 주로 하위 디렉토리들이 차지하는 공간의 용량을 계산할 때 사용

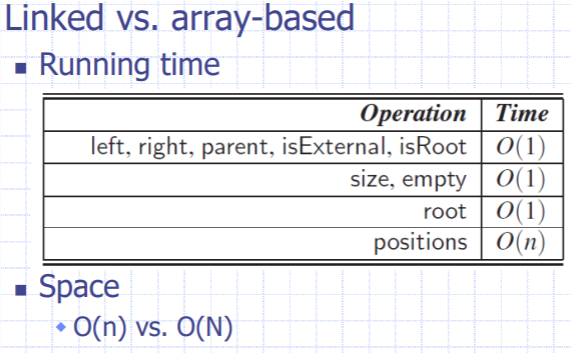
**Binary Tree**

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



* 응용: 수식표현, decision tree, searching(binary search tree)
* ADT method: Tree ADT의 함수들 + left(), right()
* Binary Tree의 Traversal
  1. Preorder: 부모🡪왼쪽 자식🡪오른쪽 자식 순서로 순회
  2. Postorder: 왼쪽 자식 🡪 오른쪽 자식 🡪 부모 순서로 순회
  3. Inorder: 왼쪽 자식 🡪 부모 🡪 오른쪽 자식 순서로 순회
  4. Euler Tour Traversal
     + preorder, inorder, postorder traversal을 모두 실행한다.
     + Tree 주위를 순회하며 각 node를 3번 방문(왼쪽 자식 방문 전, 왼쪽 자식 방문 후 오른쪽 자식 방문 전, 오른쪽 자식 방문 후)한다.
     + 한 노드당 하는 일의 수가 모두 3개로 일정하여 이 순회의 시간 복잡도는 O(n)

1. Linked list-based: 각 노드는 element, parent node, left child, right child로 구성
2. Array-base: 노드가 깊이 순서로 배열에 저장. Root=1, 왼쪽 자식=2\*i, 오른쪽 자식=2\*i+1
3. 성능 비교



**Priority Queue**

* 우선순위에 의해 데이터가 정렬되기 때문에 queue에 삽입되는 요소의 순서가 중요하지 않으며 데이터 속성에 따라 제거가 가능하다.
* ADT에서 Entry는 (key, value) 쌍으로 구성되며 key는 우선순위를 가리킴
* ADT method: insert(e), removeMin(), min(), size(), empty()

1. List-based Priority queue

List를 정렬된 상태로 저장 vs 정렬되지 않은 상태로 저장 수행 시간 비교

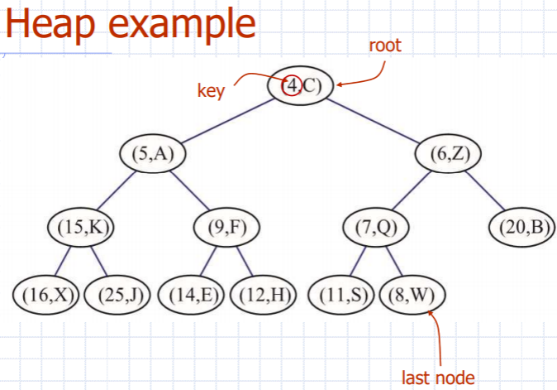
|  |  |  |
| --- | --- | --- |
| Operation | Unsorted List | Sorted List |
| Size, empty | O(1) | O(1) |
| Insert | O(1) | O(n) |
| Min, removeMin | O(n) | O(1) |

1. PQ를 이용한 정렬

* 수행 방식
  1. 배열의 값을 모두 PQ에 넣기 (insert) – n번 반복
  2. 우선순위 큐에서 값을 하나씩 기존 배열로 옮기기 (removeMin) – n번 반복
* 정렬을 위해 insert, removeMin연산을 이용하고 구현방식에 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) |

**Heap**

* 최대값, 최소값을 찾아내는 연산을 쉽게 하기 위해 고안된 자료구조
* 각 node가 (key, element)로 구성
* 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 maximum깊이의 가장 오른쪽 노드를 의미

1. 삽입 연산
2. 완전 이진 트리 구조를 유지하기 위해 삽입 시 트리의 가장 마지막에 원소를 추가한다.
3. 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.

1. 삭제 연산
2. Root key를 힙의 last node의 값으로 교체하고 last node를 제거한다.
3. 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

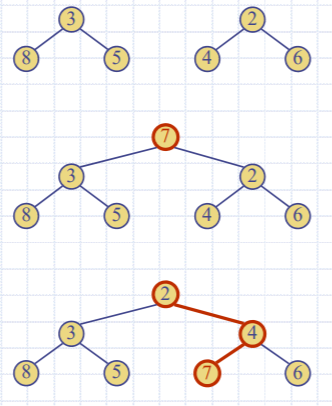
1. Heap-Sort

* Heap-based PQ: size(), empty() – O(1) time | insert, removeMin – O(log n) time
* Heap-based PQ를 이용하여 배열 정렬할 경우 O(n log n) time 걸린다.
* list기반에 PQ 방식보다 정렬 속도가 빠르다.

1. Heap 구현 방식
2. Top-Down heap 구현

* 빈 힙에 위에서부터 아래로 노드를 삽입하면서 확장하는 방식으로 insert연산을 n번 수행 (n: 노드의 개수)
* 시간 복잡도: O(n log n) time

1. Bottom-Up heap 구현

* n개의 노드가 미리 주어질 때 힙을 더 빠른 속도로 구현할 수 있는 방법
* 들어오는 키 값 순서대로 트리를 구성한 뒤, 힙 구조가 되도록 마지막 높이에서 한단계씩 높이를 올려가며 heap-order 만족여부를 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 반복하여 n개의 키를 가진 heap을 완성

**Map**

* Entry가 (key, value)쌍으로 이루어진 자료구조로 데이터 검색 시 유용하다.
* Key는 unique하다. 즉, entry별로 고유의 key를 가진다.
* ADT method: find(k), put(k, v), erase(k), size(), empty()
* 구현 방법
  1. List-based
     + Size(), empty(): O(1) time
     + Put, find, erase: worst case 경우 O(n) time (값 찾을 때 선형으로 찾으므로)
     + Map구현시 좋지 않다.
  2. Array-based
* Hash-table을 사용하는 방법이다.

**Hash Table**

* 연관배열 구조(키와 값이 1:1로 연관되어 있는 구조)를 이용하여 키에 대한 결과값을 저장하는 자료구조
* Hash function과 크기 N인 bucket array(or table)로 구성된다.

1. Hash function

* 목표: 키를 랜덤한 방법으로 가능한 충돌이 발생하지 않도록 분산하는 것
* 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 function은 hash code와 compression function을 가짐
  1. Hash code: key를 integer로 형변환, h1(k)
  2. Compression function: 정수형인 h1(k)를 [0, N-1]범위 정수값으로 매핑하는 함수 (N: 해쉬값의 분배를 randomize하기 위해 소수값으로 지정)
     + 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.

1. Separate Chaining
   * + table의 각 cell을 linked list로 구현하여 entry를 저장
     + 구현이 간단하지만 테이블이외에 추가 메모리를 사용
2. Open addressing
   * + 충돌되는 item을 테이블의 다른 cell에 저장하는 방법
     + linear probing, double hashing
3. Linear probing
   * + 충돌되는 item을 한칸씩 이동하며 다음 이용가능한 cell에 저장
     + primary clustering 문제점이 존재.
     + 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
4. 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 허용

**Binary Search Tree**

* Search Tree: ordered maps과 ordered dictionaries를 구현하는데 사용되는 tree 자료구조
* Search Tree의 ADT method: find(k), put(k, v), erase(k), size(), empty()
* Binary Search Tree
  1. 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 자료구조
  2. external node에는 값을 저장하지 않음
* 삽입 연산
  1. Root부터 leaf노드까지 k에 대한 탐색을 수행한다.
  2. K가 존재하면 값을 변경하고 존재하지 않으면 삽입
* 삭제 연산
  1. root부터 leaf노드까지 k에대한 탐색을 수행하여 해당 노드를 삭제
  2. leaf 자식 노드를 가진 경우 해당 노드와 leaf 자식 노드를 둘 다 지움
  3. 자식 노드가 모두 internal 노드인 경우, ①오른쪽 자식 중에서 가장 작은 값을 해당 위치로 옮기기 ②왼쪽 자식 중에서 가장 큰 값을 해당 위치로 옮기기 방법이 있으며 둘중 ①을 더 많이 쓴다.
* 성능
  1. n개의 item을 가진 ordered map을 높이가 h인 binary search tree로 구현 시
  2. 공간 복잡도 O(n)
  3. 삽입, 삭제, 찾기 연산의 시간 복잡도 O(h) time
  4. Worst case: h=n, best case: h=log n

**AVL Tree**

* 균형잡힌 이진 검색 트리로 모든 노드의 자식 노드간 높이 차이가 1 이하이다.
* 이진 검색 트리 중 노드 n개가 모두 한쪽으로 쏠려 높이가 n이 되는 경우같이 노드들이 한쪽으로 쏠리는 것을 방지
* AVL tree의 높이는 O(log n)
* 삽입 연산
  1. 이진 검색 트리와 마찬가지로 리프노드를 확장하는 방식으로 수행된다. 단, 삽입 후 Restructuring과정을 통해 트리의 균형을 유지(변경노드부터 root까지 중 균형깨진 첫 노드 찾기)
  2. Restructuring: Single Rotation(1번 회전), Double Rotation(2번 회전)이 존재

|  |  |
| --- | --- |
| Single Rotation | Double Rotation |
| LL(left to left): 우회전  RR(right to right): 좌회전 | LR(left to right): 두번째 노드 우회전 후, 전체 자회전  RL(right to left): 두번째 노드 좌회전 후, 전체 우회전 |
|  |  |

* 제거 연산
  1. 이진 검색 트리와 같은 방식으로 수행. 제거된 노드의 부모부터 root까지 중 불균형 생기는 노드 반복해서 찾으며 restructuring수행.
* 성능: size(), empty() – O(1) time | find(), insert(), erase() – O(log n) time

**Graph**

* 정점과 간선으로 이루어지며 트리와 달리 계층(부모-자식)이 존재하지 않는 자료구조
* 용여: 정점(노드), 간선, degree, path, cycle
* 표현방법: 간선의 정보 저장박식에 따라 인접리스트, 인접행렬 표현으로 나뉨
  1. 인접리스트: 공간복잡도 O(V+E), 간선의 수 적은 그래프에서 유리
  2. 인접행렬: 공간복잡도 O(V\*V), 두 정점간 연결 유무를 바로 알수 있고 간선 수 많은 그래프 유리

**DFS(Depth-First Search)**

* 모든 정점들을 깊이 우선으로 탐색하는 알고리즘. 그래프의 구조 알 수 있다. (componenet 개수 등)
* 시간복잡도: 인접리스트 O(V+E) , 인접행렬 O(V\*V) |

**BFS(Breadth-First Search)**

* 시작점에 가까운 정점부터 단계별로 탐색하는 알고리즘. 깊이를 알 수 있으므로 최단경로 문제를 풀 때 유용
* 시간복잡도: 인접리스트 O(V+E), 인접행렬 O(V\*V)

**TopologicalSort(위상정렬)**

* 순서가 정해져있는 작업을 차례로 수행해야 할 때 그 순서를 결정해주기 위해 사용하는 알고리즘.
* 그래프는 DAG(사이클이 없는 방향 그래프)모양을 가진다. 탐색시 노드에 번호를 부여
* 시간복잡도: O(V+E)
* 구현
  1. DFS 이용: 깊이 들어갔다가 나오면서 번호 부여 n🡪1
  2. STACK이용: 각 정점별 indegree정보를 이용해 indegree가 0인 정점부터 시작(stack에 넣기), 앞쪽부터 번호 부여 1🡪n , bfs와 유사

**Dijkstra**

* 최단 경로를 찾는 알고리즘, 가중치 있는 그래프에서 사용하는 방법이다.
* 시간복잡도: O(E log V)

**MST(Minium Spanning Tree, 최소 신장 트리)**

* 가중치 있는 그래프의 spanning tree중 간선의 가중치 합이 최소인 트리, 네트워크(가중치 그래프)에 있는 모든 정점들을 가장 적은 수의 간선과 비용으로 연결하는 트리(통신 네트워크에서 응용됨)
* 시간복잡도 : O(E log V) time
* Tree: 한 그래프내에 모든 노드들이 연결되어있고 사이클이 없는 무방향 그래프를 의미. 루트가 있으면 rooted tree
* Forest: 사이클이 없는 무방향 그래프로 한 그래프에 1개 이상의 tree들이 존재
* Spanning Tree: 그래프 내에 모든 정점을 포함하는 트리로 그래프의 최소 연결(간선) 부분 그래프이다. DFS와 BFS에서 탐색 도중 사용된 간선들과 모든 노드를 합쳐 spanning tree만듬.
* Kruskal MST 알고리즘: 탐욕방법을 이용해 MST구하는 방법. 간선들을 가중치의 오름차순으로 정렬 후, 각 단계에서 사이클을 이루지 않는 최소 비용 간선을 선택한다.
* Prim MST 알고리즘: 시작 정점에서부터 출발해 spanning tree 집합을 단계적으로 확장해나가는 방법. 정점 선택을 기반으로 한다. mst집합에 인접한 정점들 중 최소 간선 연결된 정점 선택

**Divdie-and-Conquer(분할정복)**

* 알고리즘 설계기법 중 한가지로 3가지로 나눠서 푸는 방법.
* 1. Divde(문제를 subproblem으로 나누기) 2. Conquer(재귀 or 반복으로 문제 풀기) 3. Combine(subproblem의 해를 합쳐서 원래 해를 찾기
* 이를 기반으로하는 정렬 알고리즘: merge-sort, quick-sort

**Merge-Sort** (시간복잡도: O(n log n))

크기가 n인 배열을 크기가 절반인 배열 2개로 분할하여 부분 배열을 정렬한 뒤, 두 정렬된 부분 배열을 합병하는 알고리즘. 분할 시 크기가 1이 될때까지 나눈 뒤, 거슬러 올라가며 합하면서 정렬시키고 이를 반복하며 크기가 n인 배열 완성

**Quick-Sort** (시간복잡도: worst-case 경우 O(n\*n), 평균 경우 O(n log n))

크기가 n인 배열에서 랜덤한 값 x를 골라 이 값을 기준으로 보다 작은 값들 L, 같은 값들 E, 보다 큰값들 G로 배열을 나눈뒤 L,R배열에 대해 같은 방식을 수행하며 크기가 1 or 0이 될때까지 나눈다. 이후 거슬러 올라가며 L,E,G를 합쳐 크기가 n인 정렬된 배열을 만드는 알고리즘.

**DP**

큰 문제를 작은 문제로 나눠서 푸는 기법으로 subproblem에 대해 계산된 값을 저장해서 또다시 계산하는 경우를 막는 알고리즘이다. 계산 값을 저장하므로 시간 단축