<<자료구조>>

!) [ ]안의 method들은 ADT의 대표적인 method들로 STL과는 다를 수 있음. 강의자료 기준임.

1. Linked list
2. Singly linked list - 사이에 간선이 단방향으로 하나씩만 존재.

[method: empty, front, addFront, removeFront]

1. Doubly linked list node - 사이에 간선이 양방향으로 2개씩 존재.

[method: empty, front, back, addFront, addBack, removeFront, removeBack]

1. Circular linked list - 위의 두가지 linked list에서 마지막 노드와 첫번째 노드가 연결됨.

[method: font, back, advance, add, remove]

1. Seven functions - big-O표기 같은 식들을 나타내는 7개 식
2. f(n) = c
3. f(n) = log(n)
4. f(n) = n
5. f(n) = nlog(n)
6. f(n) = n^2
7. f(n) = n^3
8. f(n) = b^n
9. 알고리즘 분석
10. Running time: best/average/worst case 별로 걸리는 시간은 분석

인풋의 사이즈에 따라 결과가 다르기 때문에 인풋의 크기를 n이라 두고 측정.

실질적인 Running time은 하드웨어나 소프트웨어의 환경에 따라 달라질 수 있기 때문에 진짜 시간이 아닌 다른 측정 방법이 필요.

이에 대한 해결책: pseudocode 상에서의 primitive operation 개수를 새는 것

* Pseudocode로 소프트웨어마다 구현방법의 차이를 없애고 primitive operation(알고리즘에서 가장 간단한 연산)의 개수를 세어 하드웨어마다 명령어 수행시간이 다른 점을 극복 가능.

1. Asymptotic algorithm analysis: 점근 분석. 위와 같은 러닝 타임이나 메모리 사용량을 big-O와 같은 점근 표기법으로 나타내어 분석하는 것.

\*Big-O: O(g(n)) <= g(n)

\*Big-Omega: Ω((g(n)) >= g(n)

\*Big-Theta: Θ(g(n)) ~ g(n)

1. Stack [method: size, empty, top, push, pop]
2. array-based: 배열크기가 꽉 찼을 때 push하면 exception throw함.

모든 함수가 O(1)에 동작함. 메모리는 처음 고정 크기 O(n)

1. linked list-based: 크기 제한이 없고 모든 함수가 O(1)에 동작, O(n)의 메모리 필요
2. Queue [method: size, empty, front, enqueue(push), dequeue(pop)]
3. array-based: 현재 front의 인덱스, 원소의 크기, 배열의 크기를 따로 지정한 변수에 저장. 이 3개의 변수들을 활용하여 고정된 배열이 끝과 처음이 연결되어 있는 것처럼 사용. O(1)시간, O(n)메모리 필요.
4. linked-list: 크기 제한 없고 모든 함수가 O(1)에 동작하고 O(n)의 메모리 필요.
5. Vector: array와 Linked list의 장점을 뽑아온 자료구조. [at, set, insert, erase, size, empty]

특정원소에 random excess가 O(1)에 가능한 linked list 라고 생각하면 됨.

모든 함수가 O(1)에 수행 가능하고 O(n)의 메모리가 필요함.

!) 미리 정해 놓은 배열 크기를 넘어섰을 때 insert 호출 시 exception을 throw하거나 더 큰 배열로 값들을 옮김.

-> Incremental strategy(상수만큼 더 큰 배열로)/Doubling strategy(2배 사이즈배열로) 값을 옮기면서 O(n)의 시간이 필요함.

1. Tree: hierarchical(계층적인) 구조를 갖는 abstract model
2. Tree에서의 용어들

-Root: 최상위 node

-Internal node: 자식이 있는 node

-External node: leaf

-Ancestors of a node: 부모를 포함한 조상 node

-Descendant of a Node: 자손 node

-Depth of Node: 조상의 수(root에서부터의 깊이

-Height of a tree: maximum of depth

-Subtree: 트리가 포함하는 트리

-Sibling: 형제 node

1. Tree 순회(탐색, traversal) 방법

- Preorder traversal: 자신을 방문하고 자식 방문.

- Postorder traversal: 자식을 모두 방문하고 자신을 방문.

1. Binary Tree: Tree중에서 모든 node의 자식이 2개 이하인 tree
2. Binary Tree만의 추가된 순회 방법

* Inorder traversal: 왼쪽 자식 방문 -> 자신 방문 -> 오른쪽 자식 방문.

1. array based, linked list based 모두 가능. 같은 성능을 가짐.

[left, right, parent, isExternal, isRoot, size, empty, root: O(1), positions: O(n)]

1. Complete(prefect) binary tree

0부터 높이 h까지의 i에 대하여 depth가 i인 노드의 개수는 2^i개

1. Almost complete binary tree

모든 리프 노드들은 왼쪽에서 오른쪽방향으로 채워지고 높이를 h라고 할 때, 0부터 h-1까지의 i에 대하여 depth가 i인 노드의 개수는 2^i개

1. Priority Queue [insert, removeMin, min, size, empty]
2. Implementation with an unsorted list

* removeMin이나 Min을 호출할 때마다 정렬.
* -> insert: O(1), removeMin: O(n), Min: O(n)

1. Implementation with a sorted list

* Insert를 할 때마다 정렬
* -> insert: O(n), removeMin: O(1), Min: O(1)

1. PQ를 사용하는 sort 방법들.
2. Selection-Sort: unsorted list를 사용하여 O(n^2)에 동작가능.

{1} Priority queue를 사용하면 n개의 원소를 insert: 1\*n

{2} 하나씩 원소를 Min을 사용하여 꺼내는 작업을 n번 수행: 1+2+…+n

=> n+1+2+3+…+n = O(n^2)

실제 selection sort는 주어진 배열에서 0부터 n-1까지의 i에 대하여 [i, n]구간에 대하여 가장 최솟값을 맨 앞에 세우는 과정을 반복.

1. Insertion-Sort: sorted list를 사용하여 O(n^2)에 동작가능.

{1} Priority queue를 사용하면 n개의 원소를 insert: 1+2+…+n

{2} 하나씩 원소를 Min을 사용하여 꺼내는 작업을 n번 수행: 1\*n

=> n+1+2+3+…+n = O(n^2)

실제 insertion sort는 배열을 차례대로 보면서 현재 보고 있는 값보다 작은 값이 나올 경우 그 값을 다시 배열의 처음부터 비교하면서 위치를 찾는 과정을 반복.

1. Heap: priority queue와 유사하지만 binary tree구조를 가짐.
2. Heap의 properties
3. Heap-Order: key(v) >= key(parent(v)) 항상 자식이 큰 값.
4. (Almost) Complete Binary Tree
5. Height of Heap: h = low bound(log n)
6. insert – Upheap: O(log n)

트리의 구조에 따라 채워져야 할 부분에 값을 넣고 부모 노드와 값보다 값이 작으면 swap하는 과정을 반복.

1. removeMin – Downheap: O(log n)

루트 노드 값이 가장 작은 값이므로 루트 값을 빼낸 후 리프 노드 들 중 가장 오른 쪽에 있는 노드(마지막 노드)를 루트의 자리에 넣는다. 그 후 downheap 과정으로 자식보다 마지막 노드가 클 경우 swap하는 과정을 반복한다.

1. Heap을 만드는 방법
2. Heap-Sort

Insert와 removeMin 과정이 O(log n) 걸리고 이 과정을 n개의 원소에 대하여 진행하므로 총 O(nlog n)의 시간이 걸린다.

1. Bottom-up heap

Heap 두개를 합치는 과정을 반복하여 heap을 완성하는 방법. 마찬가지로 O(nlog n)의 시간이 걸리지만 5)heap-sort보다 더 빠르다.

1. Map: key-value가 pair로 존재하는 abstract model

[find, put, erase, size, empty]

!) 원래의 map은 다중키를 지원하지 않았음. (STL에서는 multimap을 지원함.)

1. Map 구현방법
2. Simple list-based map

단순히 linked list로 key와 value를 entry로 한 list. put은 O(1)의 시간이 걸리고 find, erase는 O(n)걸린다.

1. Hash functions and Hash tables

Hash function에 key값을 넣어서 index를 정하는 방법.

h1 : keys → integers

h2 : integers → [0, N - 1] ex) h2(y) = (ay+b) mod N, h2(y) = |y| mod N

h(x) = h2 (h1 (x))

충돌 handling도 당연히 필요.

{1} linear probing: 충돌이 날 경우 일정한 숫자만큼 증가시켜서 저장.

{2} double hashing: 2개의 hash function을 만들어 둔 후 1번째 hash function에서의 값이 충돌 나면 첫번째 + 두번째 값을 더해서 사용.

결론: hash를 사용한다고 해도 insert, remove과정은 worst case에는 O(n)만큼의 시간이 걸릴 수 있음. 하지만 전체적으로 O(1)만큼 걸린다고 여길 수 있음. 적은 메모리를 사용할 수 있어서 compiler나 browser cache등에 사용됨.

1. Dictionary: map과 마찬가지로 key-value가 pair로 존재하는 abstract model

[find, put, erase, size, empty]

DNS mapping이나 credit card 승인(?) 용도로 사용.

Dictionary는 multiple key를 지원함. (STL에서의 multimap과 같음.)

1. 구현 방법
2. List-based: map에서와 같은 방법. Put에는 O(1)이 걸리고 find, erase에는 O(n)이 걸린다.
3. Hash Table: map과 마찬가지. 같은 key값이 존재하므로 value들의 저장 방법에 차이를 둬야함.
4. Search Table: key값을 정렬한 search table을 만드는 방법. (STL의 방법)

이 방법에서 binary search를 사용하여 key값을 검색하도록 함.

1. Binary Search Tree
2. Property

: key(왼쪽 자식) <= key(본인) <= key(오른쪽 자식)

-> Inorder traversal을 하게 되면 sort된 key들을 얻을 수 있음.

1. Search

루트에서부터 아래쪽으로 크기를 비교하면서 key값을 찾음.

1. Insertion

루트에서부터 값을 내려가면서 비교하여 삽입할 위치를 찾음.

1. Deletion

특정 key를 지우기 위해서는 그 key값의 부모와 자식들을 비교하여 자식을 올리는 과정을 반복.

1. 성능

보통 tree의 높이가 h라고 할 때 best case에서 find, put, erase의 수행시간은 O(log n)만큼 걸림.

하지만 Tree의 구조에 따라서 다른 성능을 가짐. 편향된 tree일 경우 worst case에서 find, put, erase 모두 O(n)만큼 걸릴 수 있음.

1. AVL Tree(균형 잡힌 tree)

: 모든 부모 노드에 대하여 자식 tree의 높이가 모두 1초과로 차이 나지 않는 트리

Splay tree, Red-black tree 등이 AVL tree에 속한다.

1. Height

AVL 트리의 높이가 h라고 할 때 가질 수 있는 최소 노드의 수를 n(h) 라고 하자.

n(h) = 1+n(h-1)+n(h-2) n(h) > 2^i n(h-2i)

AVL 트리에서는 위의 두 식이 성립하게 된다. 이에 따라

h <= 2log n(h) + 2 <= 2log n + 2

* Height of AVL tree는 O(log n)이 된다.

1. Insertion

Binary search tree와 같이 leaf노드에 먼저 값을 삽입한 뒤 trinode restructuring 과정으로 AVL 트리의 조건을 만족하도록 트리를 변형시킨다. (이 과정은 알고리즘 시간에 red black tree에서 더 자세하게 배우게 되므로 그때 좀 더 자세하게 언급하겠음.)

1. Remove

Remove 과정 역시 해당 노드를 지운 후 rebalancing으로 모양을 교정한다. (이 과정 역시 알고리즘 요약에서 red black tree로 좀 더 자세히 언급할 예정.)

1. Performance

Find: O(log n) put: O(log n) erase: O(log n)

1. Graph: 정점과 간선으로 이루어진 자료구조. G(V,E)로 주로 표현한다.
2. 그래프에서의 용어
3. Directed edge: 방향이 있는 간선
4. Undirected edge: 방향이 없는 간선(양방향 간선)
5. Adjacent vertex: 인접한 정점. 간선 하나로 도달할 수 있는 정점.
6. Degree of a vertex: 정점에서 연결된 간선의 수
7. Edges incident on a vertex: 정점에서 연결된 간선들.
8. Parallel edge: 중복된 간선
9. Self-loop: source(시작 정점)와 destination(도착 정점)이 같은 간선
10. Simple graph: self-loop와 parallel edge가 없는 그래프
11. Simple path: 사이클을 포함하지 않는 path(같은 정점이나 간선을 다시 방문하지 않는 path)
12. Simple cycle: 같은 정점이나 간선을 방문하지 않는 사이클
13. Subgraph: 그래프 안에 속해 있는 그래프
14. 트리도 그래프의 한 종류임.
15. 트리가 여러 개면 forest임
16. Spanning subgraph: 모든 정점들이 연결된 subgraph
17. Spanning tree: 모든 정점이 연결된 tree (forest가 아님.)
18. Directed Graph(Digraph): 간선에 방향이 있는 graph
19. Directed acyclic graph(DAG): 사이클 없는 digraph
20. Strong Connectivity: digraph에서 모든 정점에서 다른 모든 정점으로 가는 path가 존재.
21. Strong Connected Component(SCC): digraph의 subgraph로 component에 속한 모든 정점에서 다른 모든 정점으로 갈 수 있음.
22. Transitive Closure: 주어진 그래프에서 모든 정점에 대하여 다른 정점으로 직접적으로 갈 수 있는 간선을 추가한 그래프.
23. Weighted Graph: 간선에 가중치가 있는 그래프.
24. Graph ADT

[insertVertex, insertEdge, eraseVertex, eraseEdge]

1. 구현 방법
2. Adjacency list structure: 인접 리스트 방식
3. Edge list structure: 간선과 정점 구조체들이 서로 포인터로 연결
4. Adjacency Matrix structure: 인접행렬 방식.
5. Depth-First Search (DFS)

알고리즘 과정은 모두 알 테니까 생략.

인접리스트로 구현: O(n+m) 시간 소요

인접행렬로 구현: O(n^2) 시간 소요

1. Breadth-First Search (BFS)

마찬가지로 알고리즘 과정은 생략.

인접리스트로 구현: O(n+m) 시간 소요

인접행렬로 구현: O(n^2) 시간 소요

1. Topological sort(위상 정렬)

위상 순서(Topological order)에 따라 정점을 정렬하는 알고리즘.

하나의 digraph에서는 여러 개의 위상 정렬이 가능할 수 있음.

1. 들어오는 간선이 0개인 정점들을 선택하여 큐에 넣는다.
2. 큐에 있는 정점을 뺀 후 이 정점에서 나가는 간선을 모두 지운다.
3. 이 과정을 모든 정점이 큐에서 빠져나갈 때까지 반복. 큐에서 나간 순서가 위상 순서.
4. Dijkstra’s Algorithm

이 알고리즘 역시 설명은 생략하겠음.

Priority queue 또는 heap을 사용하여 최단경로가 구해진 정점에서 연결된 간선을 먼저 탐색하도록 하는 알고리즘.

인접리스트로 구현하면 O((n+m)log n)의 시간이 걸린다.

1. Minimum Spanning Tree(MST)

이름 그래도 spanning tree중에 가중치의 합이 가장 작은 tree

1. Kruskal’s Algorithm: 대표적인 MST 구하는 알고리즘

Greedy method(탐욕적 기법)을 사용하여 가중치가 작은 간선부터 선택하여 MST를 구한다. O((n+m)log n)의 시간이 걸린다.

1. 그래프의 간선들을 가중치의 오름차순으로 정렬한다.
2. 정렬된 간선 리스트에서 순서대로 사이클을 형성하지 않는 간선을 선택한다.

즉, 가장 낮은 가중치를 먼저 선택한다.

사이클을 형성하는 간선을 제외한다.

1. 해당 간선의 두 정점이 연결되어 있지 않다면 현재의 MST(최소 비용 신장 트리)의 집합에 추가한다.
2. Prim-Jarnik’s Algorithm

시작 정점에서부터 출발하여 MST를 확대해가는 방식

1. 시작 단계에서는 시작 정점만이 MST(최소 비용 신장 트리) 집합에 포함된다.
2. 앞 단계에서 만들어진 MST 집합에 인접한 정점들 중에서 최소 간선으로 연결된 정점을 선택하여 트리를 확장한다. (가장 낮은 가중치를 먼저 선택한다.)
3. 앞의 과정을 트리가 N-1개의 간선을 가질 때까지 반복한다.

앞의 (2)의 과정에서 간선을 구하는 방법에 따라 O(n^2), O((n+m)log n)으로 나뉜다.

배열로 체크하여 모두 다 봐야하는 경우 O(n^2)만큼 걸리고 간선의 정보를 heap에 넣어두고 사용하게 되면 O((n+m)log n)만큼 걸린다.

1. Merge Sort

알고리즘에서 더 자세히 다룰 예정으로 생략한다.

1. Quick sort

Merge sort와 마찬가지로 생략한다.