**자료구조**

****

**선형 자료구조**

1. **LinkedList 와 ArrayList**

**☞ Array와 ArrayList 차이**

Array는 크기를 지정해줘야 한다.

ArrayList는 리스트의 데이터로 배열을 사용하는 List 인터페이스 구현체

배열의 초기 크기를 지정할 수 있고 지정하지 않으면 기본 배열 크기는 10

원소가 가득차면 자동으로 더 큰 배열을 재할당(시간과 메모리 소모)

따라서, 크기가 큰 컬렉션을 이용할 거라면 기본 크기를 크게 잡는 것이 좋다.

|  |  |  |
| --- | --- | --- |
|  | **ArrayList** | **LinkedList** |
| **장점** | 1. 구현이 간단하다  2. 연결된 메모리상에 존재하므로 중간 데이터에 빠르게 접근 가능( 인덱스를 이용한 무작위 접근 가능)  3. 순차 접근의 경우에도 하나의 연속된 메모리 공간에 할당하므로 연결 리스트 보다 빠르다.  4. 참조를 위한 추가적인 메모리 할당이 필요 없다. | 1. 필요할 때마다 메모리를 동적으로 할당하기 때문에 효율적  2. 데이터의 삽입 삭제가 간편하다( 이전데이터와 다음데이터에 대한 참조만 변경)  3. 메모리의 재사용이 가능하다 ( 자료의 삭제시 해당 노드의 참조가 사라지므로 나중에 Garbage Collector에 의해 가용 메모리로 전환) |
| **단점** | 1. 크기를 고정해야 하기 때문에 사용될 것을 고려하여 할당해야 하고 이는 불필요한 메모리를 차지  2. 중간 데이터의 삽입 또는 삭제가 일어나면 데이터를 전부 변경해야 한다  3. 메모리의 재사용이 불가능 ( 삭제한 데이터라고 하더라도 메모리를 점유) | 1. 구현이 복잡하다  2. 중간의 데이터에 접근하기 위해서는 이전 노드를 모두 거쳐야만 접근할 수 있다.  3. 참조를 위한 추가 메모리 할당 필요 |

**성능표**

|  |  |  |
| --- | --- | --- |
|  | **ArrayList** | **LinkedList** |
| Indexing | Θ(1) | Θ(n) |
| Insert/delete at beginning | Θ(n) | Θ(1) |
| Insert/delete at end | Θ(1) | Θ(n)-last element is unknown Θ(1)-last element is known |
| Insert/delete in middle | Θ(n) | search time + Θ(1) |
| Wasted space (average) | Θ(n) | Θ(n) |

**LinkedList란?**

각 데이터 요소들에 이전데이터나 다음데이터를 가리키는 참조를 추가적으로 부여하여 자료들을 연결하는 방법

이러한 데이터 형태를 노드(node)라고 부른다. 즉 연결 리스트는 노드와 노드의 연결

**LikedList 종류**

1. 단일 연결 리스트(Singly Linked List) – 노드가 다음노드를 가리킨다.
2. 이중 연결 리스트(Doubly Linked List) – 각 노드들이 이전과 다음노드를 가리킨다.
3. 다중 연결 리스트(Multiply Linked List) – 각 노드가 2개 이상의 노드를 가리킨다.
4. 원형 연결 리스트(Circular Linked List) – 리스트의 마지막 노드가 첫번째 노드를 가리킨다.

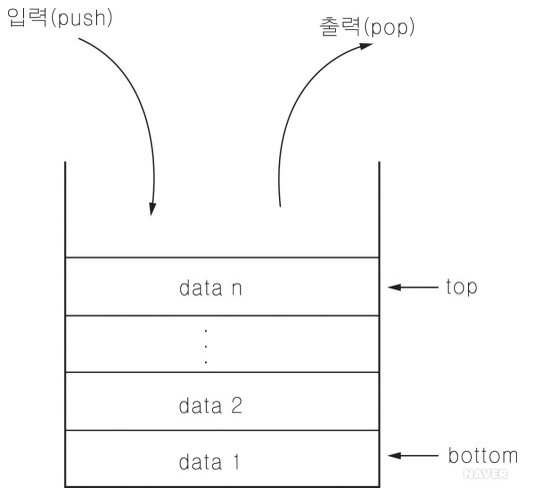
\* 저장할 데이터의 최대 개수가 정해져 있고 리스트의 중간에 데이터를 삽입, 삭제하는 작업이 많지 않으며 인덱스를 이용한 빠른 검색이 필요할 경우에는 배열

\* 저장될 데이터의 개수가 정해져 있지 않고 리스트의 중간에 데이터를 삽입하거나 삭제하는 작업이 많고 삽입, 삭제에 비해 특정 위치의 데이터를 검색하는 작업이 많지 않을 경우 연결 리스트

1. **스택(Stack) , 큐(queue) , 덱(Deque) – Restricted Structure**

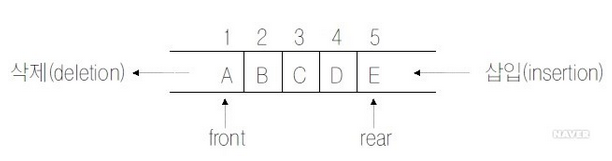
**# 스택(stack)**

**모든 원소들의 삽입(insert)과 삭제(delete)가 리스트의 한쪽 끝에서만 수행되는 제한 조건을 가지는 선형**[**자료 구조**](http://terms.naver.com/entry.nhn?docId=850544&ref=y)**(linear data structure)**로서, 삽입과 삭제가 일어나는 리스트의 끝을 top이라 하고, 다른 한쪽 끝을 bottom이라 한다. 스택은 종종 pushdown stack이라고도 하는데, 스택의 top에 새로운 원소를 삽입하는 것을 push라 하고, 가장 최근에 삽입된 원소를 의미하는 스택의 top으로부터 한 원소를 제거하는 것(**FILO**, First In Last Out / **LIFO**, Last In First Out)을 [pop](http://terms.naver.com/entry.nhn?docId=833063&ref=y)이라 한다. 이와 같은 스택 연상은 항상 스택의 top에서 발생하므로 top [포인터](http://terms.naver.com/entry.nhn?docId=832951&ref=y)의 값을 1씩 증가 또는 감소시킴으로써 수행된다.

****

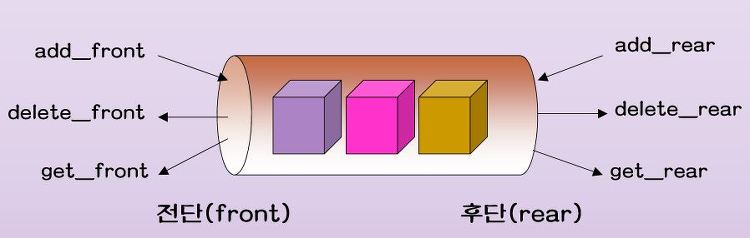
**# 큐(queue)**

리스트의 한쪽 끝에서만 삽입과 삭제가 일어나는 스택과는 달리 리스트의 한쪽 끝에서는 원소들이 삭제되고 반대쪽 끝에서는 원소들의 삽입만 가능하게 만든 [순서화](http://terms.naver.com/entry.nhn?docId=851387&ref=y)된 리스트. 가장 먼저 리스트에 삽입된 원소가 가장 먼저 삭제되므로 [선입선출](http://terms.naver.com/entry.nhn?docId=852241&ref=y)(先入先出)인 FIFO(first in first out) 리스트라고 한다.

****

**# 덱(Deque – Double Ended Queue )**

Queue 인터페이스의 확장이며 자료구조의 양 끝에 원소를 추가하고 삭제할 수 있다.

****

**시간 복잡도 – 추가, 제거, 확인 모두 O(1)**

**Stack 응용 사례**

수식의 괄호 쌍, 전위/중위/후위/표기법, DFS, Flood Fill, 문자열 뒤집기(for문이 더빠름)

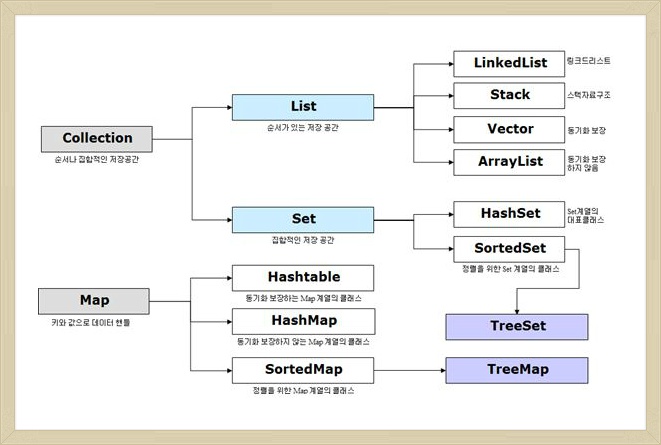
**Queue 응용 사례**

BFS, Flood Fill

**Deque 응용 사례**

양쪽에서 삽입/삭제가 일어나는 상황

1. **Java Collection FrameWork (JCF)**

****

* **Set Interface**

집합을 정의하며 요소의 중복을 허용하지 않음

* **HashSet**

가장 빠른 임의 접근 속도

순서 예측 불가

* **LinkedHashSet**

추가된 순서 또는 가장 최근에 접근한 순서대로 접근 가능

* **TreeSet**

정렬된 순서대로 보관하며 정렬방법 지정할 수 있음

* **Map Interface**

Key와 Value의 쌍으로 연관지어 저장하는 객체

* **HashMap**

Map 인터페이스를 구현하기 위해 HashTable을 사용한 클래스

중복을 허용하지 않고 순서를 보장하지 않음

키와 값으로 null이 허용

* **HashTable**

HashMap보다 느리지만 동기화 지원

키와 값으로 null이 허용되지 않음

* **TreeMap**

이진 검색 트리의 형태로 키와 값의 쌍으로 이루어진 데이터를 저장

정렬된 순서로 키, 값 쌍을 저장하므로 빠른 검색이 가능

저장 시 정렬을 하기 때문에 저장시간이 다소 오래걸림

* **LinkedHashMap**

기본적으로 HashMap을 상속받아 HashMap과 매우 흡사

Map에 있는 엔트리들의 연결 리스트가 유지되므로 입력한 순서대로 반복 가능

**Hash 란?**

<http://blog.daum.net/_blog/BlogTypeView.do?blogid=0JHcJ&articleno=8382649&categoryId=791705&regdt=20100728175237>

해쉬(Hash)는 특별한 알고리즘을 이용하여 데이터의 고유한 숫자값을 만들어 인덱스로 사용하는데 이 알고리즘을 구현한 메소드를 Hash Method라고 하며 Hash Method에 의해 반환된 데이터 고유의 숫자 값을 Hash Code 라고 한다. Hash가 내부적으로 사용하는 배열을 Hash Table이라고 하며 그 크기에 따라 성능차이가 많이 날 수 있다.

**해시함수(hash function)**란 데이터의 효율적 관리를 목적으로 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수

이 때 매핑 전 원래 데이터의 값을 **키(key)**, 매핑 후 데이터의 값을 **해시값(hash value)**, 매핑하는 과정 자체를 **해싱(hashing)**이라고 한다.

해시함수는 해쉬값의 개수보다 대개 많은 키값을 해쉬값으로 변환(many-to-one 대응)하기 때문에 해시함수가 서로 다른 두 개의 키에 대해 동일한 해시값을 내는 **해시충돌(collision)**이 발생하게 된다. 해시충돌이 발생할 가능성이 있음에도 해시테이블을 쓰는 이유는 적은 리소스로 많은 데이터를 효율적으로 관리하기 위해서

Hash Mehtod를 구형하는 방법은 여러가지가 있지만 가장 간단한 방법으로는 나머지 연산자를 이용하는 것이다.

Ex) “a”, “b”, “c” 의 hashCode가 97, 98, 99이고 Hash Table의 크기가 10이라고 했을 때 테이블에 저장될 인덱스는 다음과 같다.

97 % 10 = 7 / 98 % 10 = 8 / 99 % 10 = 9

즉, Hash Table 인덱스 7에는 “a” , 8에는 “b”, 9에는 “c”를 저장하는 방식

인덱스가 중복되면 충돌발생 / 충돌 해결방법 – 개방주소법 / 분리연결법

**분리 연결법(Chaining)** – Table 셀마다 연결리스트 하나씩 저장하도록 하고 충돌 발생 시 연결리스트의 다음 노드로 계속 추가해 가는 방식 / 데이터 검색할 때에는 Hash Table의 인덱스를 찾은 후 셀에 연결된 리스트를 순차적으로 탐색하며 찾으려는 hashCode와 저장된 노드의 hashCode를 비교

해쉬 함수 작성 가이드

* 인접키를 널리 퍼트려라 ex) 키값을 제곱, 세제곱… / 비트연산으로 비트구성을 다르게 변환
* 빈도가 높은 키영역을 넓게 펴라 ex) 입력자료의 특성 분석 필요
* 해쉬테이블의 크기는 소수(Prime Number)로 하는 것이 좋다.

**장점**

내부적으로 배열을 사용하여 데이터를 저장하기 때문에 빠른 검색 속도를 갖는다.

해시함수는 언제나 동일한 해시값을 리턴하고, 해당 색인만 알면 해시테이블의 크기에 상관없이 데이터에 빠르게 접근할 수 있으며, 색인은 계산이 간단한 함수(상수시간)로 작동. 다시 말해 데이터 액세스(삽입, 삭제, 탐색)시 O(1) 지향

해시는 보안 분야에서도 널리 사용. 키와 해시값 사이에 직접적인 연관이 없기 때문에 해시값만 가지고는 키를 온전히 복원하기 어렵기 때문. 아울러 해시함수는 길이가 서로 다른 입력데이터에 대해 일정한 길이의 출력을 만들 수 있어서 ‘데이터 축약’ 기능도 수행할 수 있다.

**☞ TreeMap과 HashMap의 차이**

TreeMap에서는 컬렉션이 순서대로 저장되므로 전체 컬렉션을 반복해서 순회할 때 키의 순서가 보존된다.

HashMap 클래스에서는 순서가 보존되지 않는다. 키들이 객체의 hashCode 메서드 값에 따라 저장되기 때문

주로 HashMap을 사용하고, 정렬이나 범위검색이 필요한 경우에만 TreeMap을 사용하는 것이 좋다.