# 자료구조 실습05

# Data Structures Lab05

- **Doubly Linked List 정의 및 응용**
- **Iterator Class 정의 및 응용**
- **프로그램과제 힌트**

# DoublySortedLinkedList ADT

```
template <typename T>
class DoublySortedLinkedList
{
    friend class DoublyIterator<T>;
public:
                    DoublySortedLinkedList();          // Default constructor
                    ~DoublySortedLinkedList();             // Destructor

                    bool IsFull();                         // 리스트가 가득 찼는지 확인
                    void MakeEmpty();                  // 리스트를 비움
                    int GetLength() const;                 // 리스트가 보유하고 있는 item 개수 반환
                void Add(T item);                  // 새로운 레코드 추가
                void Delete (T item);                  // 기존 레코드 삭제
                void Replace (T item);                 // 기존 레코드 갱신
                    int Get(T &item);                      // Primary key를 기준으로 데이터를 검색하고
해당 데이터를 가져옴

private:
                DoublyNodeType<T> *m_pFirst;       // 리스트의 처음 노드를 가리키는 포인터
                DoublyNodeType<T> *m_pLast;        // 리스트의 마지막 노드를 가리키는 포인터
                int m_nLength;                     // 리스트에 저장된 레코드 수
}
```

# DoublySortedLinkedList

```
template <typename T>
struct DoublyNodeType
{
    T data;                         // data
    DoublyNodeType *prev;           // Pointer of previous node
    DoublyNodeType *next;           // Pointer of next node
}
```

# Doubly Linked List를 위한 Iterator Class

```
template <typename T>
class DoublyIterator
{
     friend class DoublySortedLinkedList<T>;
     public:
          DoublyIterator(const DoublySortedLinkedList<T> &list) : m_List(list),
          m_pCurPointer(list.m_pFirst) {};
          // set the current pointer to the first node
          void Begin() {
                    m_pCurPointer = m_List.m_pFirst->next;
          }
          // move current pointer to next
          void Next() {
                    m_pCurPointer = m_pCurPointer->next;
          }
          // current node의 data 포인터를 리턴
          T* GetCurrentNode() {
                    return &m_pCurPointer->data;
          }
          // not end?
          bool NotEnd() {
               if (m_pCurPointer->next == NULL)
                  return false;
               else
                  return true;
          }

     private:
          const DoublySortedLinkedList<T> &m_List;
          DoublyNodeType<T>* m_pCurPointer;
};
```

# DoublySortedLinkedList 주요 member functions

```
// 리스트를 초기화하여 생성
template <typename T>
DoublySortedLinkedList<T> :: DoublySortedLinkedList()
{
    // header와 trailer 역할을 할 2개의 dummy node를 생성
    m_pFirst = new DoublyNodeType<T>;
    m_pLast = new DoublyNodeType<T>;
    // 두 노드가 서로를 가리키도록 연결
    m_pFirst->next = m_pLast; // 끝과 처음이 서로를 가리키게 초기화.
    m_pLast->prev = m_pFirst; // 끝과 처음이 서로를 가리키게 초기화.
    // 앞과 뒤는 null로 초기화
    m_pLast->next = NULL;
    m_pFirst->prev = NULL;
    m_nLength = 0; // 길이는 0.
}
```

# DoublyLinkedList Member 함수

```
template <typename T>
int DoublySortedLinkedList<T>::Get(T &item)
{
   DoublyIterator<T> itor(*this);
   itor.Begin();
   while (itor.m_pCurPointer != m_pLast)
   {
      if (itor.m_pCurPointer->data == item)
      {
          item = itor.m_pCurPointer->data;
          return 1; // 일치하면 1을 반환.
      }
      else
          itor.Next(); // 다음으로 이동.
   }
   return 0; // 일치하지 않으면 0을 반환.
}
```

```
template <typename T>
int DoublySortedLinkedList<T>::Get(T &item)
{
   DoublyIterator<T> itor(*this);
   for (itor.Begin(); itor.NotEnd(); itor.Next())
   {
      if (*itor.GetCurrentNode() == item)
      {
          item = *itor.GetCurrentNode();
          return 1; // 일치하면 1을 반환.
      }
   }

   return 0; // 일치하지 않으면 0을 반환.
}
```

# DoublyLinkedList Member 함수

```
template <typename T>
int DoublySortedLinkedList<T>::Add(T item)
{
     DoublyIterator<T> itor(*this);
     itor.Begin(); // 다음으로 이동.
     for (itor.Begin(); itor.NotEnd(); itor.Next())
     {
          if (item < itor.m_pCurPointer->data) // 현재 노드가 크면 현재 노드 앞에 추가
                    break;  // exit the loop
          else if (item == itor.m_pCurPointer->data) // 동일 Primary key존재
                    return 0; // return with 0
          itor.Next(); // 다음으로 이동.
     }
     // end with m_pCurPointer==m_pLast or item<m_pCurPointer->data
     // 두경우 모두 현재 노드 앞에 추가
     DoublyNodeType<T> *pItem = new DoublyNodeType<T>;
     pItem->data = item;
     pItem->prev = itor.m_pCurPointer->prev;
     pItem->next = itor.m_pCurPointer;
     itor.m_pCurPointer->prev->next = pItem;
     itor.m_pCurPointer->prev = pItem; // 아이템을 삽입.
     m_nLength++;
     return 1;
}
```

# Application Class

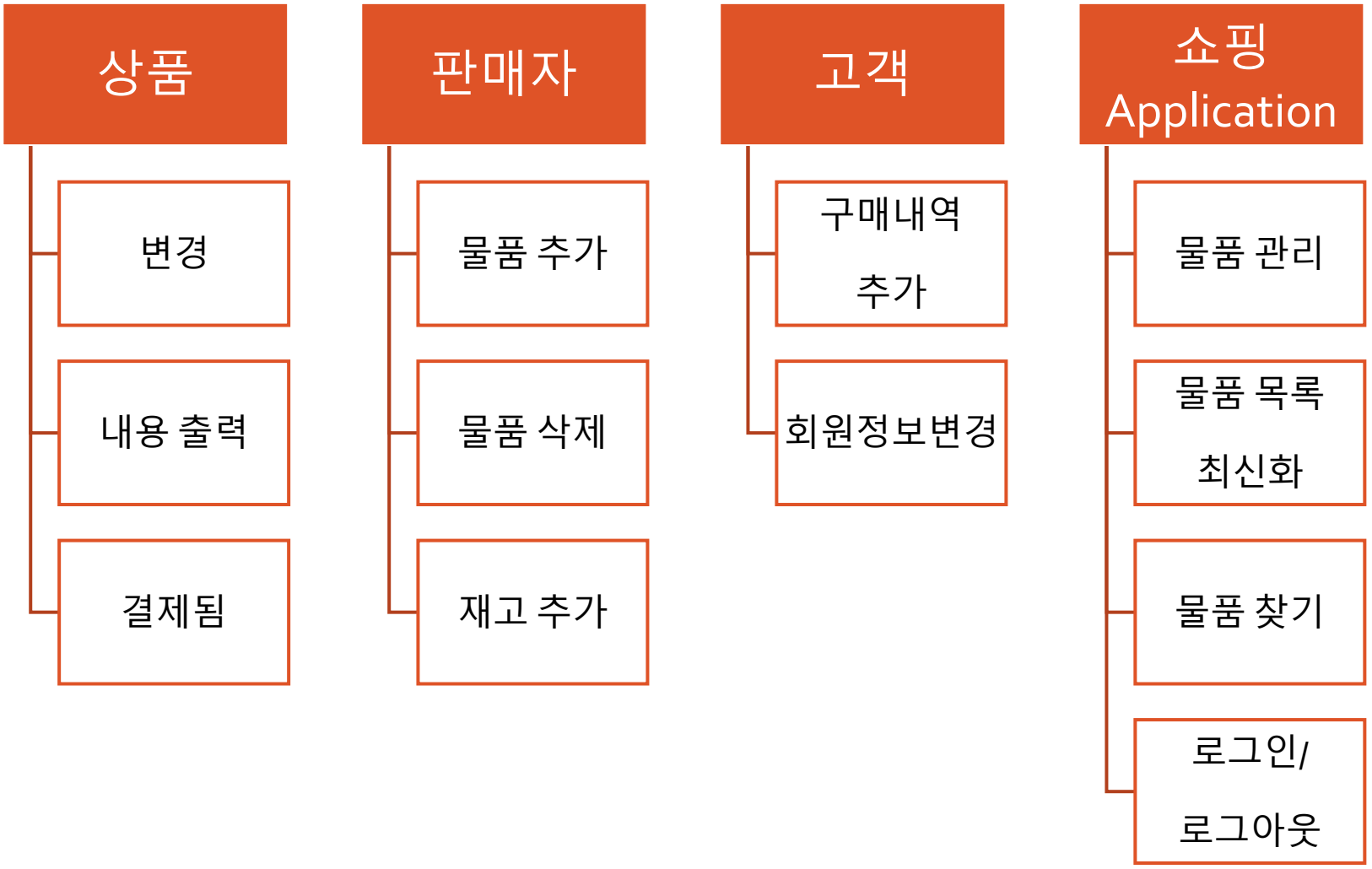
```
class Application{
public:
    Application(){ m_Command = 0;}
    ~Application(){}
    void Run();
    int GetCommand();
    int AddItem();
    void DisplayAllItem();
    int OpenInFile(char *fileName);
    int OpenOutFile(char *fileName);
    int ReadDataFromFile();
    int WriteDataToFile();
    void MakeEmptyList();
    int DeleteItem();
    int UpdateInfo();
    int SearchItembyID();
    void SearchByName();
private:
    ifstream m_InFile;///< 입력 파일 디스크립터.
    ofstream m_OutFile;///< 출력 파일 디스크립터.
    DoublySortedLinkedList<ItemType>m_List; ///< 아이템 리스트.
    int m_Command;///< 현재 command 숫자.
};
```

# 실습 5 과제

- 실습 4까지 구현한 쇼핑몰 관리 시스템의 기본적인 기능은 다음과 같다.
  - ✓ 쇼핑몰에서 등록된 모든 상품들의 자세한 정보를 저장하는 master list 정의 및 관련 기능 구현
  - ✓ 하나의 장바구니를 관리하기 위한 BasketType정의 및 관련 기능 구현
  - ✓ 말단 카테고리에 저장된 상품들을 관리하기 위한 CateType 정의 및 관련된 기능 구현
- 실습 5 과제에서는 다음 기능들을 추가한다.
  - Singly linked list로 구현한 카테고리의 상품 리스트를 Header와 trailer node를 이용한 doubly linked list로 구현한다.
  - 기존 list class에 내재된 iteration을 독립적인 iterator class를 분리하고 기존 코드에서 사용된 iteration 함수를 새로 정의한 iteration class를 이용하여 구현한다.
  - Doubly linked list나 동적으로 할당된 배열을 이용하여 계층적인 구조의 카테고리 메뉴 기능을 구현
- 과제 제출
  - 이 과제는 프로그램 1과 동일하다. 따라서 프로그램 1을 제출하면 실습 5제출한 것으로 간주함

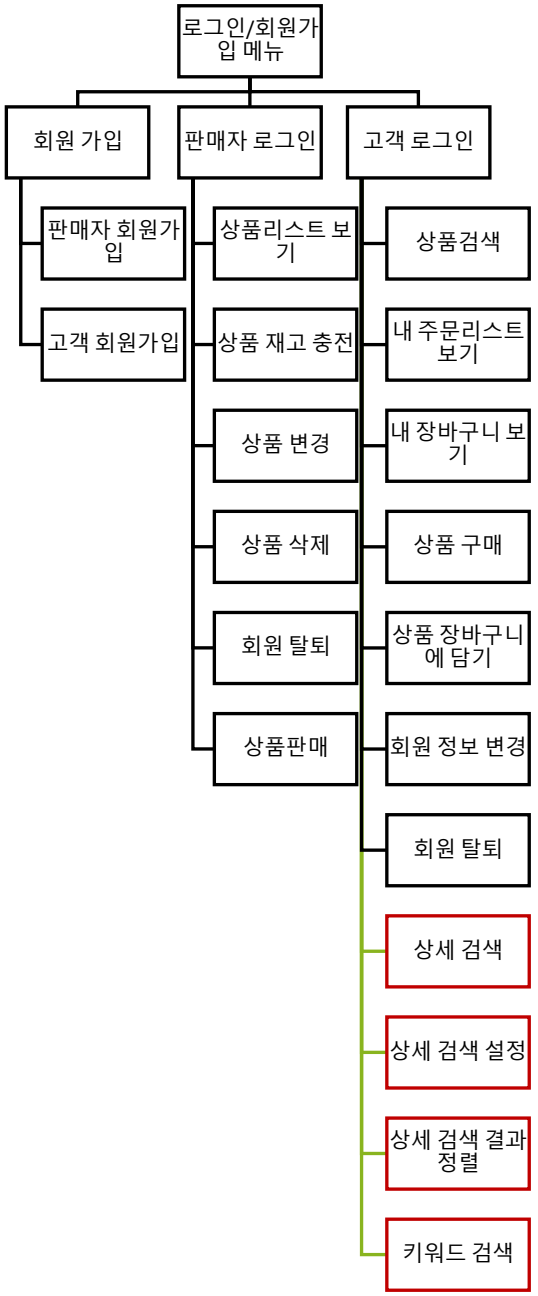
# 이전 학생 자료: 기능

## 학생 1



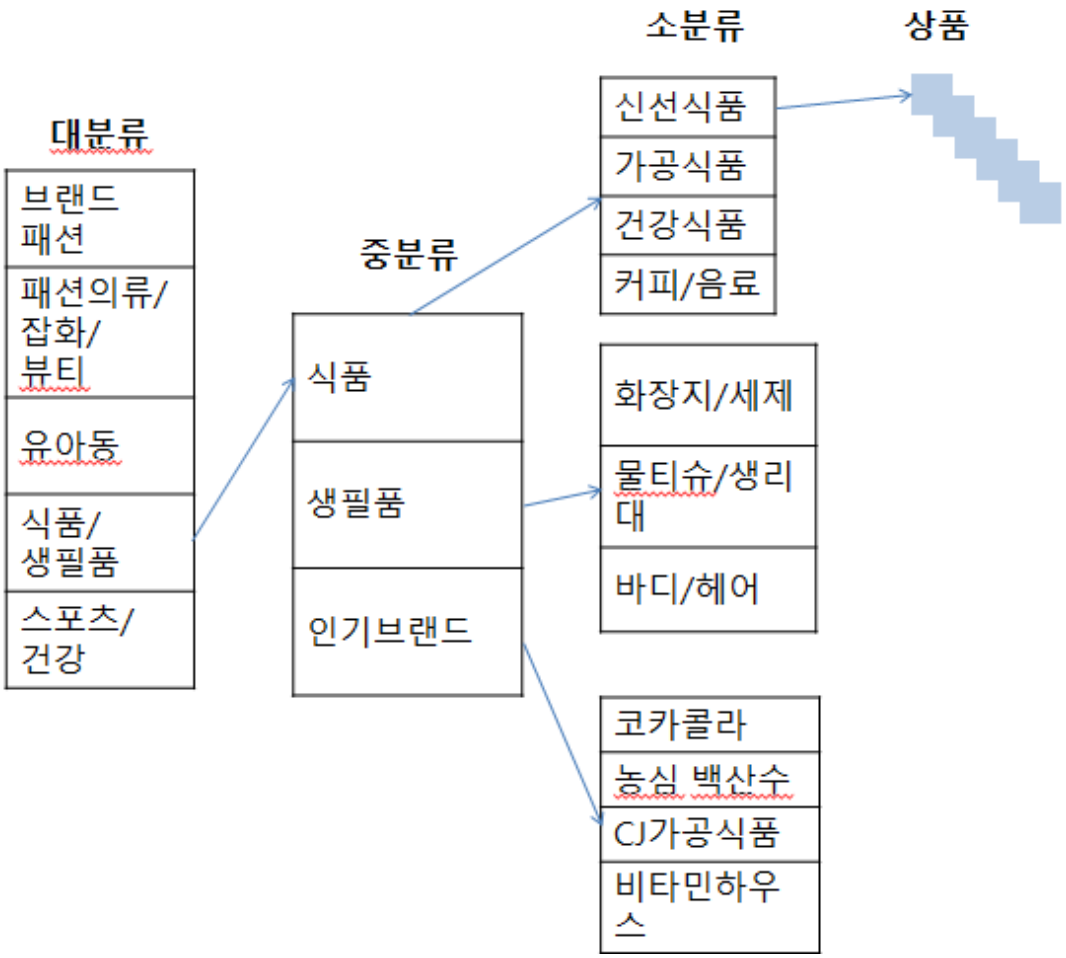
# 이전 학생 자료: 기능

## 학생 1

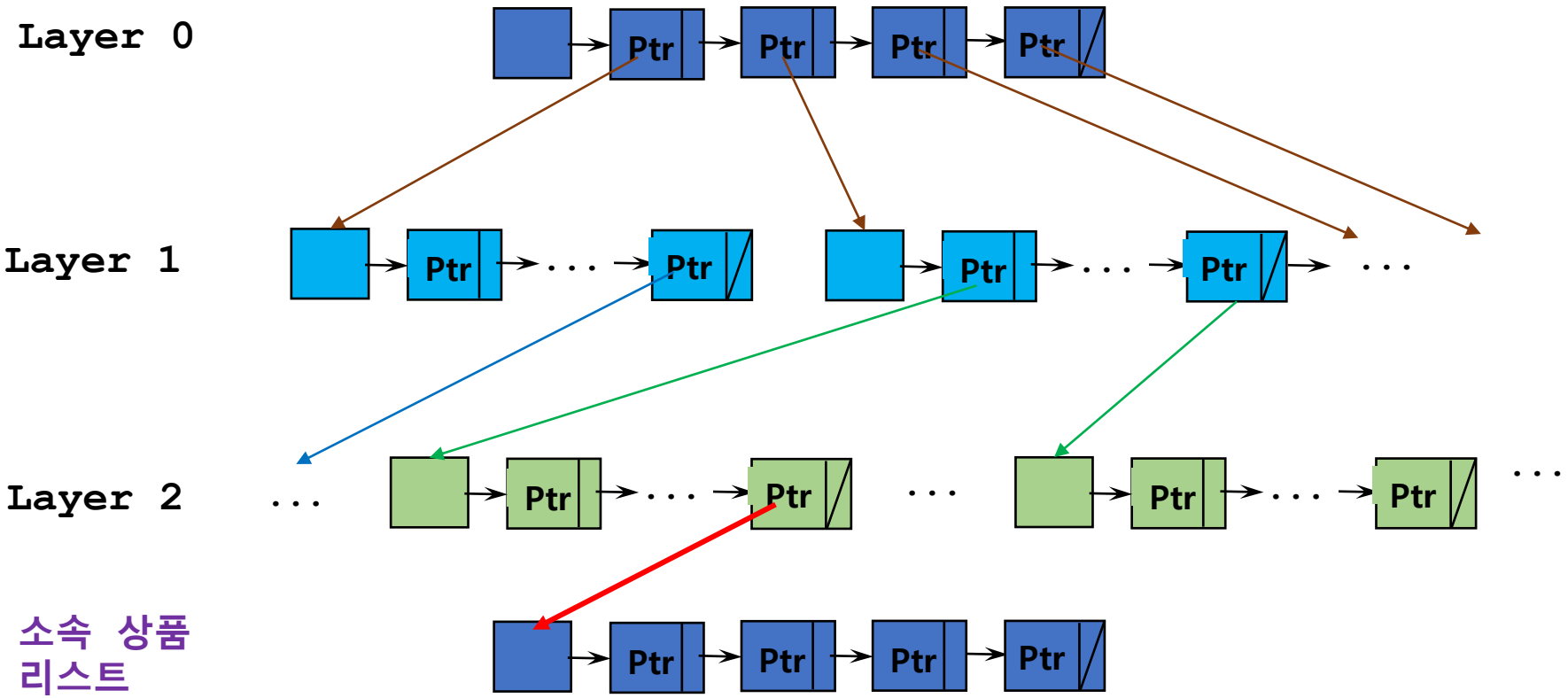


# 이전 학생 자료: ADT
## 학생 1

# 이전 학생 자료: 카테고리 메뉴
## 학생 1

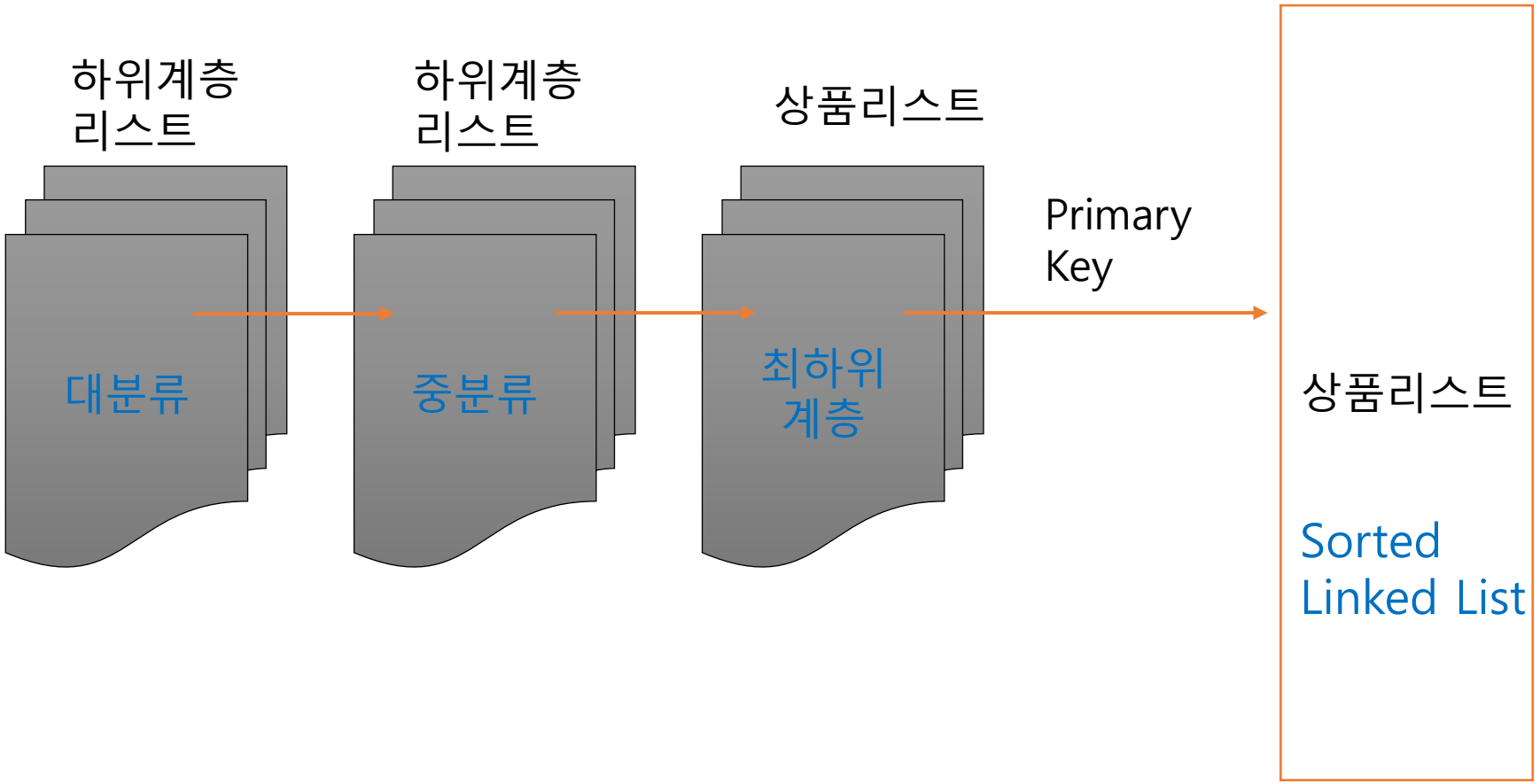


# 카테고리 메뉴

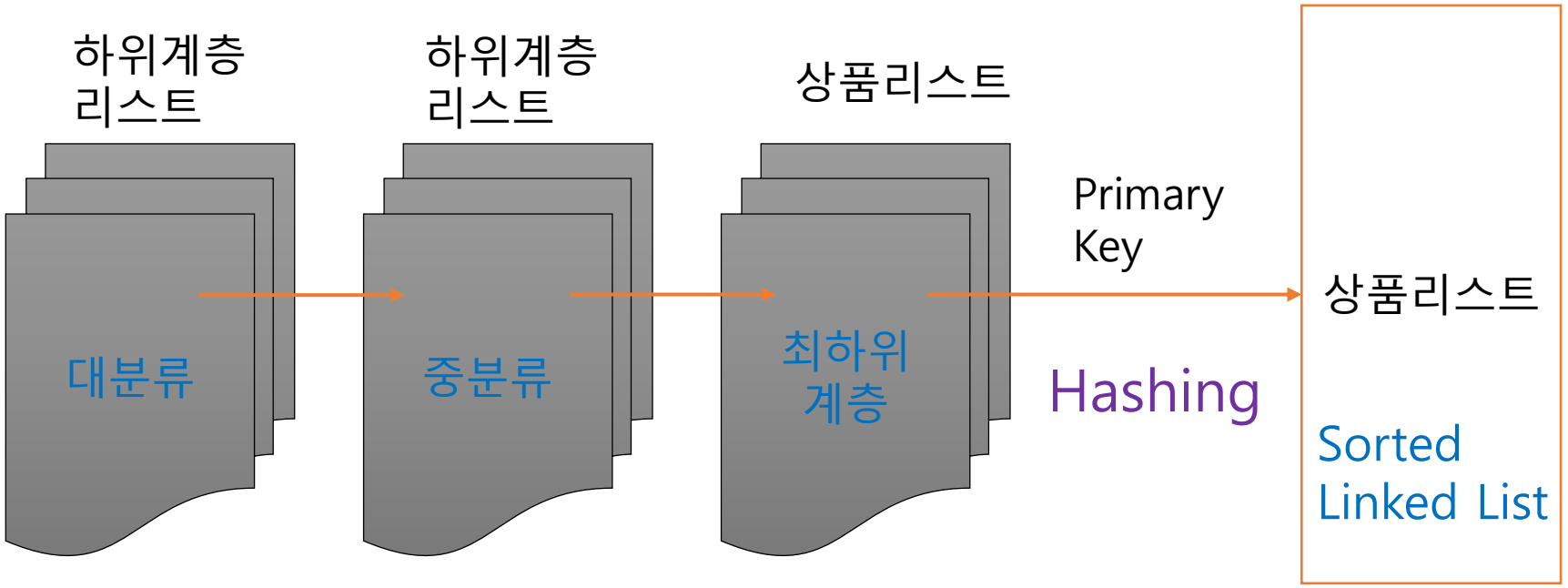



- 카테고리 계층구조의 설계는 Linked List나 배열을 이용할 수 있음
- 하지만 재귀적인 메모리 할당을 방지하기 위해서는 각 노드에 저장할 하위계층 정보는 포인터를 이용

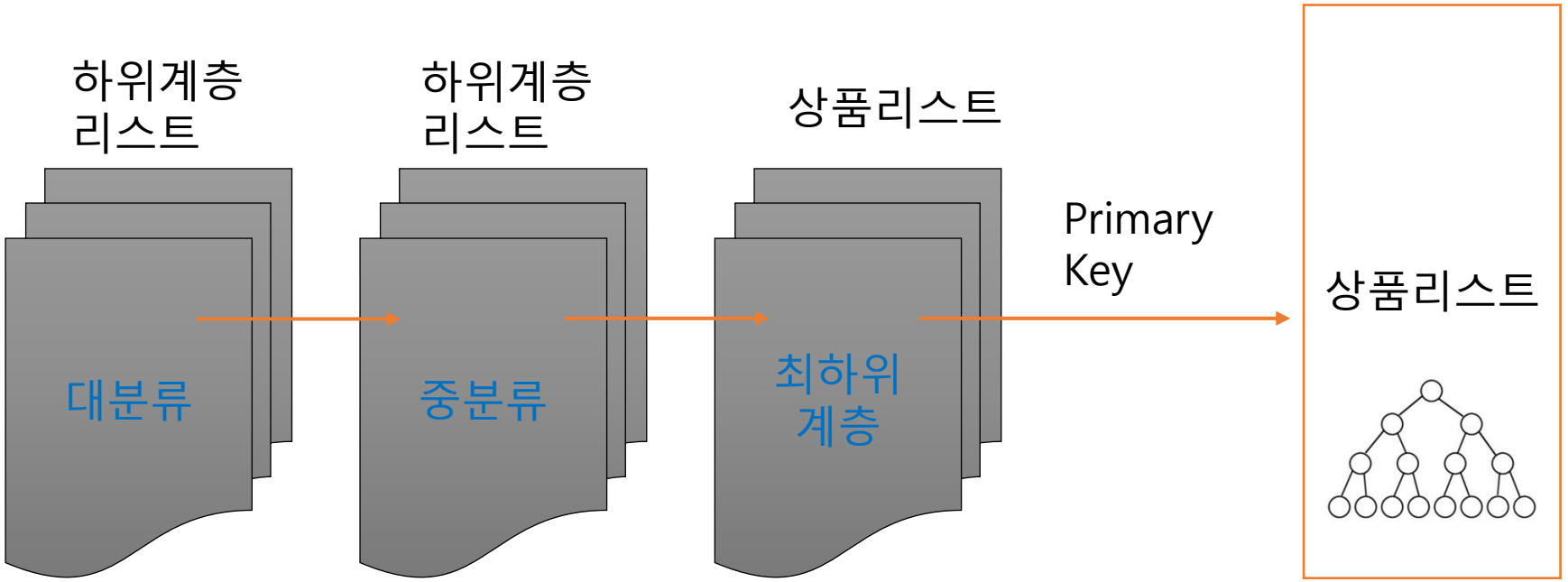
# 전체적인 자료구조-Case 1



# 전체적인 자료구조-Case 2



# 전체적인 자료구조-Case 3




검색이 빠른 Tree
또는 배열을 이용