* **컴퓨터의 문제 해결 과정**

****

* **자료구조**란 자료를 효율적으로 사용하기 위해서 자료의 특성에 따라서 분류하여 구성하고 저장 및 처리하는 모든 작업.
* **자료구조를 배워야하는 이유**는 컴퓨터가 효율적으로 문제를 처리하기위해서는 문제를 정의하고 분석하여 그에 대한 최적의 프로그램을 작성해야 하기 때문이다.
* **단순 구조**

정수, 실수, 문자, 문자열 등의 기본 자료형.

* **선형 구조**

자료들 간의 앞뒤 관계가 1:1의 선형 관계.  
Ex) 배열, 연결 리스트, 스택, 큐, 덱 등.

* **비선형 구조**

자료들 간의 앞뒤 관계가 ‘1:다’ 또는 ‘다:다’의 관계.  
Ex) 트리 그래프 등.

* **파일 구조**

레코드의 집합인 파일에 대한 구조.  
Ex) 순차 파일, 색인 파일, 직접 파일 등.

* **배열**

같은 자료형을 가진 자료들을 나열하여 메모리에 연속으로 저장된 자료들의 그룹.

*인덱스*(index)는 배열의 요소를 간단히 구별하기 위해 사용되는 번호.

int i[2][3] = {{1, 2, 3}, {4, 5, 6}};  
int i[2][3] = {1, 2, 3, 4, 5, 6};  
int i[][3] = {{1, 2, 3}, {4, 5, 6}};

* **포인터**

변수의 메모리 주소값. 포인터 변수는 주소값을 저장하는 특별한 변수다.

* **구조체**

구조체도 배열처럼 여러 개의 데이터를 그룹으로 묶어서 하나의 자료형으로 정의하고 사용하는 자료형이다.

배열은 같은 자료형 만을 그룹으로 묶을 수 있지만, 구조체는 서로 다른 자료형을 그룹으로 묶을 수 있으므로 복잡한 자료 형태를 정의하는데 유용하게 사용된다.

*레코드*(record)는 자료를 체계적으로 관리하기 위해서 구성한 일정한 단위 형신이다. 여러 개의 레코드가 모여서 *파일*(file)을 구성한다. *필드*(field)는 레코드를 구성하는 하위 항목이다.

* **순차 자료구조 (Sequential Data Structure)**

자료의 논리적인 순서와 물리적인 순서가 일치하는 자료구조. Ex) 배열.

* **연결 자료구조 (Linked Data Structure)**

자료의 논리적인 순서와 물리적인 순서가 일치하지 않는 자료구조. Ex) 연결 리스트 (Linked List).

각 원소에 저장되어 있는 다음 원소의 주소에 의해 순서가 연결되는 방식. 물리적인 순서를 맞추기 위한 오버헤드가 발생하지 않음.

여러 개의 작은 공간을 연결하여 하나의 전체 자료구조를 표현. 크기 변경이 유연하고 더 효율적으로 메모리를 사용.

* **순차 자료구조의 단점**

삽입 또는 삭제 연산 후, 연속적인 물리 주소를 유지하기 위해서 원소들을 이동시키는 추가적인 작업과 시간이 소요됨. 즉, 배열이 갖고 있는 메모리 사용의 비효율성 문제를 그대로 가짐.

* **단순 연결 리스트**

첫 번째 노드로 삽입

|  |
| --- |
| insertFirstNode(L, x)  new := getNode();  new.data := x;  new.link := L;  L := new;  end insertFirstNode() |

중간 노드로 삽입

|  |
| --- |
| insertMiddleNode(L, pre, x)  new := getNode();  new.data := x;  if(L = null) then {  L := new;  new.link := null;  } else {  new.link := pre.link;  pre.link := new;  }  end insertMiddleNode() |

마지막 노드로 삽입

|  |
| --- |
| insertLastNode(L, x)  new := getNode();  new.data := x;  new.link := null;  if(L = null) then {  L := new;  return;  }  temp := L;  while(temp.link != null) do {  temp := temp.link;  }  temp.link := new;  end insertLastNode() |

노드 삭제

|  |
| --- |
| deleteNode(L, pre)  if(L = null) then error;  else {  old := pre.link;  if(old = null) then return;  pre.link := old.link;  }  returnNode(old);  end deleteNode() |

노드 탐색

|  |
| --- |
| searchNode(L, x)  temp := L;  while(temp != null) do {  if(temp.data = x) then {  return temp;  }  temp := temp.link;  }  return temp;  end searchNode() |

* **스택 (Stack)**

접시를 쌓듯이 자료를 차곡차곡 쌓아 올린 형태의 자료구조. 스택에 저장된 원소는 top으로 정한 곳에서만 접근 가능. 후입선출, Last-In-First-Out (LIFO).

push

|  |
| --- |
| psuh(S, x)  top := top + 1;  if(top > STACK\_SIZE) then {  overflow;  } else {  S(top) := x;  }  end push() |

pop

|  |
| --- |
| pop(S)  if(top = 0) then{  underflow;  } else {  return S(top);  top := top -1;  }  end pop() |

Applicable for: 역순 문자열 만들기, 시스템 스택, 모바일 앱의 Activity.

* **큐 (Queue)**

스택과 마찬가지로 삽입과 삭제의 위치가 제한되어 있는 유한 순서 리스트. 큐의 뒤에서는 삽입만 하고, 앞에서는 삭제만 할 수 있는 구조. 선입선출, First-In-First-Out (FIFO).

* **원형 큐**

큐에서 삽입과 삭제를 반복하면서 앞자리가 비어 있지만 포화상태로 인식하고 더 이상의 삽입을 수행하지 않는다. 원형 큐는 논리적으로 배열의 처음과 끝이 연결되어 있다고 가정하는 것이다.

* **순차자료구조를 이용한 큐의 구현**

초기 공백 큐 생성

|  |
| --- |
| createQueue()  Q[n];  front := -1;  rear := -1;  end createQueue() |

공백 큐 검사

|  |
| --- |
| isEmpty(Q)  if(front = rear) then {  return true;  } else return false;  end isEmpty() |

포화 상태 검사

|  |
| --- |
| isFull(Q)  if(rear = QUEUE\_SIZE - 1) then {  return true;  } else return false;  end isEmpty() |

원소 삽입

|  |
| --- |
| enQueue(Q, item)  if(isFull(Q)) then {  Queue\_Full();  } else {  rear := rear + 1;  Q[rear] := item;  }  end enQueue() |

원소 반환

|  |
| --- |
| deQueue(Q)  if(isEmpty(Q)) then {  Queue\_Empty();  } else {  front := front + 1;  return Q[front];  }  end isEmpty() |

원소 삭제

|  |
| --- |
| delete(Q)  if(isEmpty(Q)) then {  Queue\_Empty();  } else front := front + 1;  end delete() |

원소 검사

|  |
| --- |
| peak(Q)  if(isEmpty(Q)) then {  Queue\_Empty();  } else return Q[front + 1];  end peak() |

* **연결자료구조를 이용한 큐의 구현**

초기 공백 연결 큐 생성

|  |
| --- |
| createLinkedQueue()  front := null;  rear := null;  end createQueue() |

공백 연결 큐 검사

|  |
| --- |
| isEmpty(LQ)  if(front = null) then {  return true;  } else return false;  end isEmpty() |

원소 삽입

|  |
| --- |
| enQueue(LQ, item)  new := getNode();  new.data := item;  new.link := null;  if(front = null) then {  rear := new;  front := new;  } else {  rear.link := new;  rear := new;  }  end enQueue() |

원소 반환

|  |
| --- |
| deQueue(LQ)  if(isEmpty(LQ)) then {  Queue\_Empty();  } else {  old := front;  item := front.data;  front := front.link;  if(isEmpty(LQ)) then {  rear := null;  }  returnNode(old);  return item;  }  end deQueue() |

원소 삭제

|  |
| --- |
| delete(LQ)  if(isEmpty(LQ)) then {  Queue\_Empty();  } else {  old := front;  front := front.link;  if(isEmpty(LQ)) then {  rear := null;  }  returnNode(old);  }  end delete() |

* **트리 (Tree)**

원소들 간에 ‘1:다’ 관계를 가지는 비선형 자료구조.  
원소들 간에 계층 관계를 가지는 계층형 자료구조.



트리 A의 *노드*(원소)는 A-L이다.  
A는 *루트 노드*(Root Node)라고 한다.  
B의 *부모 노드*(Parent Node)는 A.  
B의 *자식 노드*(Child Node)는 E, F.  
B, C, D는 서로 *형제 노드*(Sibling Node).  
K의 *조상 노드*(Ancestor Node)는 F, B, A.  
B의 *자손 노드*(Descendant Node)는 E, F, K, L.  
트리 A의 *단말 노드*(Leaf Node)는 E, K, L, G, H, I, J.

*간선*(Edge)은 노드를 연결하는 선이다. 부모 노드와 연결된 간선을 끊으면 *서브 트리*(Subtree)가 생성된다. 즉, 각 노드는 자식 노드의 개수만큼 서브 트리를 갖는다.

*노드의 차수*(Degree)는 노드에 연결된 자손 노드의 레벨 수 이다. 즉, A의 차수는 3, B의 차수는 2이다.

*트리의 차수*는 트리에 있는 노드의 차수 중 가장 큰 값이다. 즉, 트리 A의 차수는 3이다.

*높이*(Height)는 루트 노드에서 가장 깊이 있는 노드까지 가는 경로의 길이이다. 즉, 트리 A의 높이는 3이다.

*깊이*(Depth)는 루트 노드에서 특정 노드까지 가는 경로의 길이 이다. 즉, 트리 A에서 노드 F의 깊이는 2이다.

* **이진 트리 (Binary Tree)**

트리의 노드 구조를 일정하게 정의하여 트리의 구현과 연산이 쉽도록 정의한 트리. 이진 트리의 모든 노드는 왼쪽과 오른쪽 자식 노드 만을 가진다.

N개 노드를 가진 이진 트리는 항상 N-1개 간선을 가진다.

루트 노드를 제외한 모든 노드가 부모 노드와 연결되는 한 개의 간선을 가지기 때문이다.

높이가 H인 이진 트리가 가질 수 있는 노드의 최소 개수는 H+1개가 되며, 최대 개수는 개가 된다.

이진 트리의 높이가 H가 되려면 한 레벨에 최소 한 개의 노드가 있어야하므로 높이가 H인 이진 트리의 최소 노드의 개수는 H+1개.

하나의 노드는 최대 2개의 자식 노드를 가질 수 있으므로 레벨 L에서의 노드의 최대 개수는 2L개 이므로 높이가 H인 이진 트리 전체의 노드 개수는 개.

* **포화 이진 트리 (Full Binary Tree)**는 모든 레벨에 노드가 포화상태로 채워져 있는 트리.
* **완전 이진 트리 (Complete Binary Tree)**는 노드 번호 1번부터 n번까지 빈자리가 없이 연속된 트리.
* **편향 이진 트리 (Skewed Binary Tree)**는 한쪽 방향의 자식 노드 만을 가진 트리.
* **순회 (Traversal)**

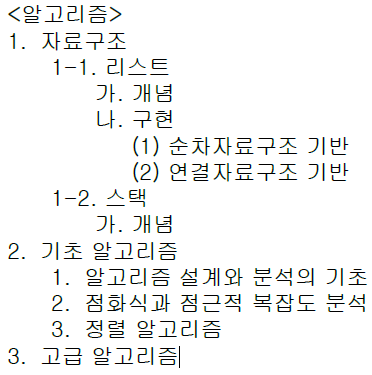
****

전위(Preorder) 순회 A-B-D-H-E-I-J-C-F-G-K

중위(Inorder) 순회 H-D-B-I-E-J-A-F-C-G-K

후위(Postorder) 순회 H-D-I-J-E-B-F-K-G-C-A

* **트리 순회의 응용**

  
*전위 순회*는 트리 구조로 저장된 목차(Index) 출력.

  
*중위 순회*는 트리를 이용한 수식 연산의 표기

  
*후위 순회*는 디렉토리의 용략 계산

* **이진 탐색 트리 (Binary Search Tree; BST)**

모든 원소는 서로 다른 유일한 키를 갖는다. 왼쪽 서브 트리 원소의 키는 그 루트의 키보다 작다. 반대로 오른쪽은 그 루트의 키보다 크다. 따라서, 왼쪽 서브 트리와 오른쪽 서브 트리도 BST다.

* **바람직한 알고리즘**은*명확해야* 해야한다. 이해하기 쉽고 가능하면 간결하게 기술하며, 지나친 기호적 표현은 오히려 명확성을 떨어뜨린다. 명확성을 해치지 않으면 일반 언어의 사용도 무방하다. 수행 시간을 고려하여 *효율적이어야* 한다.
* **알고리즘의 수행 시간**

|  |
| --- |
| sample(A[], n){  k := n/2;  return A[k];  } |
| n에 관계 없이 상수 시간 소요. |

|  |
| --- |
| sample(A[], n){  sum := 0;  for i := 1 to n  sum := sum + A[i]  return sum;  } |
| n에 비례한 시간 소요. |

|  |
| --- |
| sample(A[], n){  sum := 0;  for i := 1 to n  for j := 1 to n  sum := sum + A[i] \* A[j];  return sum;  } |
| n^2에 비례한 시간 소요. |

|  |
| --- |
| sample(A[], n){  sum := 0;  for i := 1 to n  for j := 1 to n  sum := sum + max(A)  return sum;  } |
| n^3에 비례한 시간 소요. |

|  |
| --- |
| factorial(n){  if (n = 1) return 1;  return n \* factorial(n - 1);  } |
| n에 비례한 시간 소요. |

* **점근적 분석 (Asymptotic Analysis)**

입력의 크기가 충분히 큰 경우에 대한 분석. 점근적 분석은 O(Omicron), Ω(Omega), Θ(Theta) 등을 이용하여 표기한다. 이러한 방법을 점근적 표기법(Asymptotic Notation)이라고 한다.

* **O(f(n)); Omicron; Upper Bound**

충분히 큰 모든 에 대하여 인 양의 상수 가 존재한다.

즉, 에서 는 보다 느리거나 같은 정도로 증가한다.

* **Ω(f(n)); Omega; Lower Bound**

충분히 큰 모든 에 대하여 인 양의 상수 가 존재한다.

즉, 에서 는 보다 빠르거나 같은 정도로 증가한다.

* **Θ(f(n)); Theta; Tight Bound**

즉, 에서 는 와 같은 속도로 증가한다.

* **다음 알고리즘의 시간 복잡도는**

|  |
| --- |
| sample(A[], n){  sum1 := 0;  for I := 1 to n  sum1 := sum1 + A[n];  sum2 := 0  for i := 1 to n  for j := 1 to n  sum2 := sum2 + A[i] \* A[j]  return sum1 + sum2;  } |
|  |

* **정렬 알고리즘의 복잡도**

선택 정렬 힙 정력   
퀵 정렬 , avg.

* **검색의 복잡도**

Array   
Binary Search tree , avg.   
Balanced BST   
Hash Table avg.