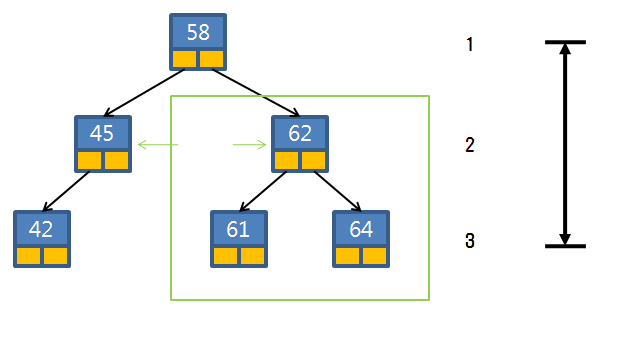
**———————————————————————————————————————————————————**

[1] 기초 문제

———————————————————————————————————————————————————

1. **다음 그림에서 찾을 수 있는 트리의 용어들에 어떤 것이 있는지 나열하시오.**



1. **트리 용어와 관련된 문제다. 빈 칸에 들어가는 단어를 맞춰보도록 한다.**
   1. ① 서브트리 ② 자식 ③ 왼쪽 ④ 오른쪽
   2. ① 루트 ② 최상위 ③ 머리
   3. ① 단말 ② 자식 ③ 마지막 ④ 꼬리
   4. ① 키 ② 이진 탐색 트리 ③ 선택
   5. ① 순회 ② 방문 ③ 전위 ④ 중위 ⑤ 후위
2. **트리를 사용하고 있는 곳을 세 가지씩만 나열해 보자.**
   1. 회사 조직, 종속과목강문계, 족보, 다단계, 토너먼트
   2. 폴더, 심볼 테이블, 결정 트리, 수식 트리, 사전

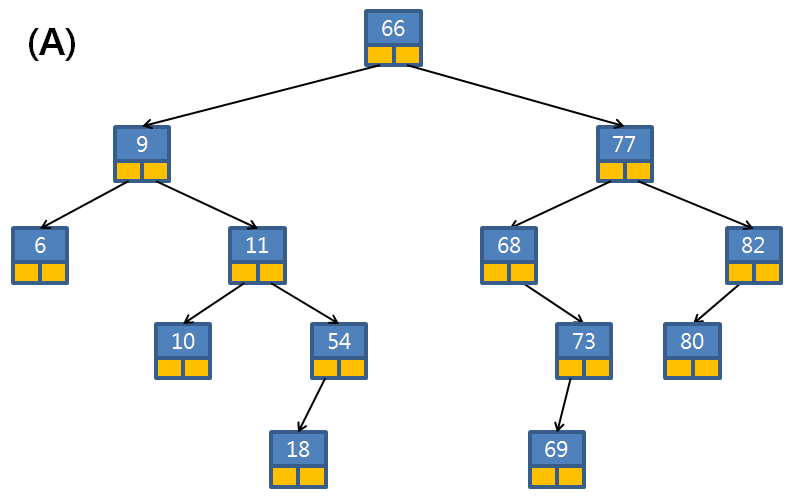
|  |
| --- |
| 해답  정답으로 간주할 수 있는 것들이 너무 많아서 이번 장에서 설명한 것만 실었습니다. |

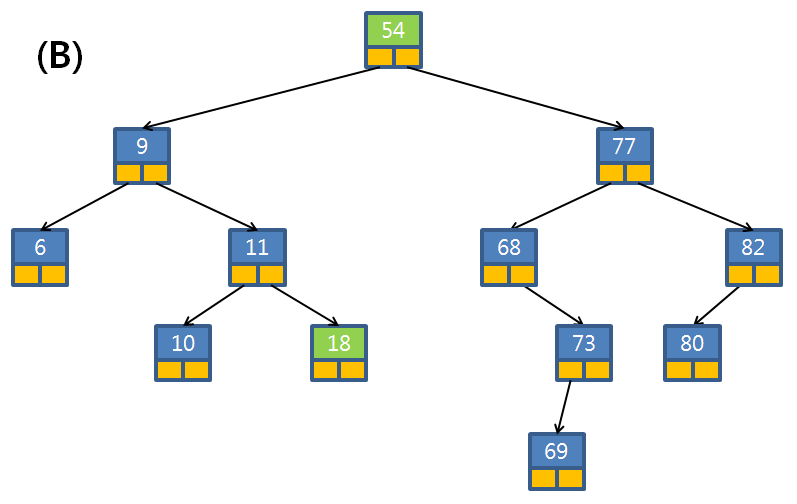
1. **이진 탐색 트리는 모든 트리를 대표할 만큼 폭넓게 사용되는 대표 트리다. 배열 또한 정렬이라는 주제를 따로 배정할 만큼 이진 탐색 트리에 비해 못할 게 없다. 정렬의 관점에서 빈 칸을 채워보자.**

|  |  |  |
| --- | --- | --- |
| 구분 | 배열 | 트리 |
| 메모리 | 충분한 크기로 일괄 할당 및 해제. 데이터만 저장하므로 트리에 비해 덜 사용한다. | 추가하거나 삭제할 때마다 필요한 만큼 할당 및 해제. 데이터와 포인터를 함께 저장하므로 배열에 비해 최소 포인터 두 개만큼 더 사용한다. |
| 반복 | 반복문을 사용하고, 배열의 처음부터 마지막까지 단순 반복을 하면 쉽게 전체 데이터를 얻을 수 있다. | 재귀 함수를 사용하고, 함수 호출에 대한 비용 때문에 배열에 비해 전체를 순회하는 비용은 상당하다. |
| 시기 | 프로그램을 시작할 때, 파일 등에서 읽어온 데이터를 정렬시켜서 사용해야 할 때 혹은 종료할 때 사용하던 데이터를 정렬시켜서 파일에 저장해야 할 때. | 데이터 추가와 삭제가 빈번하게 발생하는 과정에서 항상 정렬을 유지해야 할 때. |
| 추가/삭제 | 정렬된 이후에 추가되는 데이터에 대해서는 최악의 성능을 보장한다. 배열은 삽입과 삭제가 최대 단점이다. | 단말 노드까지 이동하는 비용밖에 발생하지 않으므로 최적의 성능을 보장한다. |

|  |
| --- |
| 해답  표에 대한 완전한 설명은 4장의 연계 학습에 있습니다. 설명이 너무 길어 기본적인 답안만 제공합니다. |

1. **이진 탐색 트리를 구성하는 문제다. 아래에 나온 20개의 숫자를 차례대로 추가했다. 여기서 다루는 트리는 중복 요소를 저장할 수 없다고 가정한다.**





1. **본문에 나왔던 코드에 대해 설명을 하는 곳이다. 트리를 배우려는 친구에게 설명한다는 생각으로 차분하게 기술해 보자.**

|  |
| --- |
| 해답  자세한 설명이 2장의 심화 학습에 있습니다. 설명이 너무 길어 답안은 생략했습니다. |

**———————————————————————————————————————————————————**

[2] 기본 실습 문제

———————————————————————————————————————————————————

1. **본문에서 배웠던 가벼운 함수들이다. 직접 구현해 보고 본문 코드와 비교해 보도록 하자.**
   1. NODE\* makeNode(int key)

{

NODE\* node = malloc(sizeof(NODE));

node->key = key;

node->left = node->right = NULL;

return node;

}

* 1. void printInOrder(NODE\* pRoot)

{

if(pRoot != NULL)

{

printInOrder(pRoot->left);

printf("%d ", pRoot->key);

printInOrder(pRoot->right);

}

}

* 1. void ChangePointer(int\*\* pp)

{

static int g = 100;

\*pp = &g;

}

1. **본문에서 배웠던 조금 무거운 함수들이다. 직접 구현해 보고 본문 코드와 비교해 보도록 하자.**
   1. int countNode(NODE\* pRoot)

{

if(pRoot == NULL)

return 0;

return 1 + countNode(pRoot->left) + countNode(pRoot->right);

}

* 1. NODE\* findKey(NODE\* pRoot, int key)

{

NODE\* cur = pRoot;

while(cur != NULL)

{

if(cur->key == key)

return cur;

if(key < cur->key) cur = cur->left;

else cur = cur->right;

}

return NULL;

}

* 1. NODE\* getPredecessor(NODE\* cur)

{

if(cur == NULL || cur->left == NULL)

return NULL;

for(cur = cur->left; cur->right; cur = cur->right)

;

return cur;

}

* 1. deleteNode 함수는 삭제 여부를 알려주기 때문에, 삭제됐을 때만 새로운 키를 갖는 노드를 추가하면 됩니다. 전체 함수를 새로 꾸밀 필요는 없어 보입니다.

int changeKey(NODE\*\* ppRoot, int oldKey, int newKey)

{

int ret = deleteNode(ppRoot, oldKey);

if(ret == 1)

insertNode(ppRoot, newKey);

return ret;

}

1. **저자는 여러분이 복잡한 코드까지 직접 구현하기를 바라지는 않는다. 대신 비어있는 곳의 코드를 채울 정도는 되어야 한다고 생각한다. 이번 장에서는 많은 종류의 코드를 다룬 것이 아니라서 deleteNode 함수만으로 구성을 했다. 빈 칸에 들어가는 코드는 무엇인가?**
   1. ① while(node)

② if(node->left && node->right)

③ if(node->left)

④ if(key < node->key)

* 1. ① while(node = \*ppRoot)

② if(key < node->key)

③ if(node->left && node->right)

④ node = \*ppRoot;

**———————————————————————————————————————————————————**

[3] 응용 실습 문제

———————————————————————————————————————————————————

1. pRoot가 NULL인지 검사하는 것은 반드시 필요합니다. 자식이 하나인 상태에서 left나 right에 대해 자신을 호출합니다.

int countLeaf(NODE\* pRoot)

{

if(pRoot == NULL)

return 0;

if(pRoot->left == NULL && pRoot->right == NULL)

{

printf("단말 : %d\n", pRoot->key);

return 1;

}

return countLeaf(pRoot->left) + countLeaf(pRoot->right);

}

아래처럼 수정할 수도 있지만, 좋아 보이진 않습니다.

int countLeaf(NODE\* pRoot)

{

if(pRoot == NULL)

return 0;

return (!pRoot->left && !pRoot->right) + countLeaf(pRoot->left) + countLeaf(pRoot->right);

}

1. void\*를 저장하는 큐를 사용했습니다. NODE\*를 저장하게 되면 큐가 트리에 종속되기 때문에 좋지 않게 생각됐습니다.

void printLevelOrder(NODE\* pRoot)

{

if(pRoot == NULL)

return;

enQueue(pRoot);

while(isEmpty() == 0)

{

NODE\* cur = deQueue();

printf("%d ", cur->key);

if(cur->left ) enQueue(cur->left );

if(cur->right) enQueue(cur->right);

}

}

1. 문제를 제시할 땐 난이도가 있다고 얘기했지만, 재귀 함수를 정확하게 이해하기만 하면 오히려 쉬운 문제입니다.

void makeSymmetry(NODE\* pRoot)

{

if(pRoot != NULL)

{

NODE\* temp;

temp = pRoot->left;

pRoot->left = pRoot->right;

pRoot->right = temp;

makeSymmetry(pRoot->left );

makeSymmetry(pRoot->right);

}

}