**알고리즘 과제3**

(Optimal Binary Search Tree 실험)

logo.gif

|  |  |
| --- | --- |
| 과목명 | 알고리즘 |
| 담당교수 | 김계영 |
| 학과 | 소프트웨어학부 |
| 학년 | 3 |
| 학번 | 20152994 |
| 성명 | 이진영 |
| 제출일 | 2017.11.13 |



**과제3 – Optimal Binary Search Tree 실험**

**(Dynamic Programming vs Divide-and-Conquer)**

20152994 이진영

**1. 원시 코드**

**1) 프로그램 1 (Dynamic Programming Method)**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define N1 3  #define N2 2  #define N3 4  // 예제 3.7  // key[1] ~ key[N}의 범위를 갖는 N개의 key  char\* tree\_key1[N1 + 1] = { "", "key1","key2","key3" };  // key[1] ~ key[N]의 각 키가 search key일 확률 key\_prob[1] ~ key\_prob[N]  double tree\_prob1[N1 + 1] = { 0, 0.7, 0.2, 0.1 };  // 예제 3.8  char\* tree\_key2[N2 + 1] = { "", "key2","key3" };  double tree\_prob2[N2 + 1] = { 0, 0.2, 0.1 };  // 예제 3.9  char\* tree\_key3[N3+1] = { "", "Don","Isabelle","Ralph","Wally" };  double tree\_prob3[N3+1] = { 0, (double)3/8, (double)3/8, (double)1/8, (double)1/8 };  typedef struct node\* nodePointer;  typedef struct node {  char\* key;  nodePointer leftChild;  nodePointer rightChild;  } node;  // 최적 이분검색트리 생성을 위한 정보를 제공하는 함수  double optSearchTree(int num, double\* prob, double\*\* avg\_t, int\*\* root);  void main() {  // 최소 평균 검색시간  double min\_avg1;  // avg\_time[i][j]: i~j의 index를 가진 key 집합의 최소 평균 검색 시간을 저장하는 배열  double\*\* avg\_time1;  // root\_of[i][j]: i~j의 index를 가진 key의 집합에서, root값으로 선정된 key의 index  int\*\* root\_of1;  avg\_time1 = (double\*\*)calloc((N1 + 2), sizeof(double\*));  root\_of1 = (int\*\*)calloc((N1 + 2), sizeof(int\*));  for (int i = 0; i < N1 + 2; i++) {  avg\_time1[i] = (double\*)calloc((N1 + 1), sizeof(double));  root\_of1[i] = (int\*)calloc((N1 + 1), sizeof(int));  }    double min\_avg2;  double\*\* avg\_time2;  int\*\* root\_of2;  avg\_time2 = (double\*\*)calloc((N2 + 2), sizeof(double\*));  root\_of2 = (int\*\*)calloc((N2 + 2), sizeof(int\*));  for (int i = 0; i < N2 + 2; i++) {  avg\_time2[i] = (double\*)calloc((N2 + 1), sizeof(double));  root\_of2[i] = (int\*)calloc((N2 + 1), sizeof(int));  }  double min\_avg3;  double\*\* avg\_time3;  int\*\* root\_of3;  avg\_time3 = (double\*\*)calloc((N3 + 2), sizeof(double\*));  root\_of3 = (int\*\*)calloc((N3 + 2), sizeof(int\*));  for (int i = 0; i < N3 + 2; i++) {  avg\_time3[i] = (double\*)calloc((N3 + 1), sizeof(double));  root\_of3[i] = (int\*)calloc((N3 + 1), sizeof(int));  }  min\_avg1 = optSearchTree(N1, tree\_prob1, avg\_time1, root\_of1);  printf("예제 3.7의 최소 평균 검색시간 : %f\n", min\_avg1);  min\_avg2 = optSearchTree(N2, tree\_prob2, avg\_time2, root\_of2);  printf("예제 3.8의 최소 평균 검색시간 : %f\n", min\_avg2);  min\_avg3 = optSearchTree(N3, tree\_prob3, avg\_time3, root\_of3);  printf("예제 3.9의 최소 평균 검색시간 : %f\n", min\_avg3);  }  double optSearchTree(int num, double\* prob, double\*\* avg\_t, int\*\* root){  int i, j, k,m, diagonal;  double min, sigma = 0;  int min\_k;  double min\_avg;  for (i = 1; i <= num; i++) {  avg\_t[i][i - 1] = 0;  avg\_t[i][i] = prob[i];  root[i][i] = i;  root[i][i - 1] = 0;  }  avg\_t[num + 1][num] = 0;  root[num + 1][num] = 0;  for(diagonal = 1; diagonal<= num-1; diagonal++)  for (i = 1; i <= num - diagonal; i++) {  j = i + diagonal;  min = avg\_t[i][i - 1] + avg\_t[i + 1][j];  min\_k = i;  for (k = i; k <= j; k++) {  if (min > avg\_t[i][k - 1] + avg\_t[k + 1][j]) {  min = avg\_t[i][k - 1] + avg\_t[k + 1][j];  min\_k = k;  }  }  sigma = 0;  for (m = i; m <= j; m++) {  sigma += prob[m];  }  avg\_t[i][j] = min + sigma;  root[i][j] = min\_k;  }  return min\_avg = avg\_t[1][num];  } |

**2) 프로그램 2 (Divide-and-Conquer (Recursive) Method)**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define N1 3  #define N2 2  #define N3 4  // 예제 3.7  // key[1] ~ key[N}의 범위를 갖는 N개의 key  char\* tree\_key1[N1 + 1] = { "", "key1","key2","key3" };  // key[1] ~ key[N]의 각 키가 search key일 확률 key\_prob[1] ~ key\_prob[N]  double tree\_prob1[N1 + 1] = { 0, 0.7, 0.2, 0.1 };  // 예제 3.8  char\* tree\_key2[N2 + 1] = { "", "key2","key3" };  double tree\_prob2[N2 + 1] = { 0, 0.2, 0.1 };  // 예제 3.9  char\* tree\_key3[N3 + 1] = { "", "Don","Isabelle","Ralph","Wally" };  double tree\_prob3[N3 + 1] = { 0, (double)3 / 8, (double)3 / 8, (double)1 / 8, (double)1 / 8 };  typedef struct node\* nodePointer;  typedef struct node {  char\* key;  nodePointer leftChild;  nodePointer rightChild;  } node;  // 최적 이분검색트리 생성을 위한 정보를 제공하는 함수  double optSearchTree(int num, double\* prob, int\*\* root);  // 재귀 함수로써, i~j의 index를 가진 key 집합의 최소 평균 검색시간과 root 배열을 return  double minAvgTime(int i, int j, double\* prob, int\*\* root);  void main() {  // 최소 평균 검색시간  double min\_avg1;  // root\_of(i,j) : i~j의 index를 가진 key의 집합에서, root값으로 선정된 key의 index  int\*\* root\_of1;  root\_of1 = (int\*\*)calloc((N1 + 2), sizeof(int\*));  for (int i = 0; i < N1 + 2; i++) {  root\_of1[i] = (int\*)calloc((N1 + 1), sizeof(int));  }  double min\_avg2;  int\*\* root\_of2;  root\_of2 = (int\*\*)calloc((N2 + 2), sizeof(int\*));  for (int i = 0; i < N2 + 2; i++) {  root\_of2[i] = (int\*)calloc((N2 + 1), sizeof(int));  }  double min\_avg3;  int\*\* root\_of3;  root\_of3 = (int\*\*)calloc((N3 + 2), sizeof(int\*));  for (int i = 0; i < N3 + 2; i++) {  root\_of3[i] = (int\*)calloc((N3 + 1), sizeof(int));  }  printf("= Divide-and-Conquer =\n");  min\_avg1 = optSearchTree(N1, tree\_prob1, root\_of1);  printf("예제 3.7의 최소 평균 검색시간 : %f\n", min\_avg1);  min\_avg2 = optSearchTree(N2, tree\_prob2, root\_of2);  printf("예제 3.8의 최소 평균 검색시간 : %f\n", min\_avg2);  min\_avg3 = optSearchTree(N3, tree\_prob3, root\_of3);  printf("예제 3.9의 최소 평균 검색시간 : %f\n", min\_avg3);  }  double optSearchTree(int num, double\* prob, int\*\* root) {  double min\_avg;  return min\_avg = minAvgTime(1, num, prob, root);  }  double minAvgTime(int i, int j, double\* prob, int\*\* root) {  double min\_avg, temp\_min\_avg;  double sigma = 0;  if (j == i - 1)  return 0;  if (i == j)  return prob[i];  for (int m = i; m <= j; m++) {  sigma += prob[m];  }  min\_avg = minAvgTime(i, i - 1, prob, root) + minAvgTime(i + 1, j, prob, root) + sigma;  root[i][j] = i;  for (int k = i; k <= j; k++) {  temp\_min\_avg = minAvgTime(i, k - 1, prob, root) + minAvgTime(k + 1, j, prob, root) + sigma;  if (min\_avg > temp\_min\_avg) {  min\_avg = temp\_min\_avg;  root[i][j] = k;  }  }  return min\_avg;  } |

**2. 실행 결과**

**1) 프로그램 1**

**① 입력 자료**

//예제 3.7

tree\_key1[N1+1] = { "", "key1","key2","key3" };

tree\_prob1[N1+1] = { 0, 0.7, 0.2, 0.1 };

double min\_avg1;

double\*\* avg\_time1; // (N1+2)x(N1+1) 행렬

int\*\* root\_of1; // (N1+2)x(N1+1) 행렬

//예제 3.8

tree\_key2[N2 + 1] = { "", "key2","key3" };

tree\_prob2[N2 + 1] = { 0, 0.2, 0.1 };

double min\_avg2;

double\*\* avg\_time2; // (N2+2)x(N2+1) 행렬

int\*\* root\_of2; // (N2+2)x(N2+1) 행렬

//예제 3.9

tree\_key3[N3 + 1] = { "", "Don","Isabelle","Ralph","Wally" };

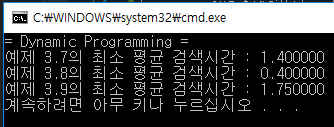
tree\_prob3[N3 + 1] = { 0, (double)3 / 8, (double)3 / 8, (double)1 / 8, (double)1 / 8 };

double min\_avg3;

double\*\* avg\_time3; // (N3+2)x(N3+1) 행렬

int\*\* root\_of3; // (N3+2)x(N3+1) 행렬

**② 실행 결과**

****

**2) 프로그램 2**

**① 입력 자료**

//예제 3.7

tree\_key1[N1+1] = { "", "key1","key2","key3" };

tree\_prob1[N1+1] = { 0, 0.7, 0.2, 0.1 };

double min\_avg1;

int\*\* root\_of1; // (N1+2)x(N1+1) 행렬

//예제 3.8

tree\_key2[N2 + 1] = { "", "key2","key3" };

tree\_prob2[N2 + 1] = { 0, 0.2, 0.1 };

double min\_avg2;

int\*\* root\_of2; // (N2+2)x(N2+1) 행렬

//예제 3.9

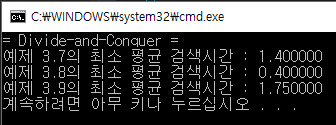
tree\_key3[N3 + 1] = { "", "Don","Isabelle","Ralph","Wally" };

tree\_prob3[N3 + 1] = { 0, (double)3 / 8, (double)3 / 8, (double)1 / 8, (double)1 / 8 };

double min\_avg3;

int\*\* root\_of3; // (N3+2)x(N3+1) 행렬

**② 실행 결과**

****

**3. 성능비교**

**① 표**

|  |  |  |
| --- | --- | --- |
|  | **Dynamic** | **Recursive** |
| 키 개수 | 단위연산 수행횟수 | 단위연산 수행횟수 |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 2 | 2 | 2 |
| 3 | 9 | 11 |
| 4 | 25 | 46 |
| 5 | 55 | 178 |
| 6 | 105 | 672 |
| 7 | 182 | 2517 |
| 8 | 294 | 9404 |
| 9 | 450 | 35108 |
| 10 | 660 | 131038 |
| 11 | 935 | 489055 |
| 12 | 1287 | 1825194 |
| 13 | 1729 | 6811734 |
| 14 | 2275 | 25421756 |
| 15 | 2940 | 94875305 |
| 16 | 3740 | 354079480 |

**② 그래프**