Algorithms and Data Structures

Francesco Ranellucci

dicember 05, 2021

Contents

1 Sorting Algorithms

1.1 Insertion Sort

O(n^2)
Vector divided in left sorted and right unsorted, when there is a number higher in the left compared to another on the right they are switched

0	1	2	3	4	5
4	2	6	3	1	5
4	(2)	6	3	1	5
2	4	6	3	1	5
2	4	(6)	3	1	5
2	4	6	3	1	5
2	4	6	(3)	1	5
2	3	4	6	1	5
2	3	4	6	(1)	5
1	2	3	4	6	(5)
1	2	3	4	5	6

```
//2 subarray
//left is sorted, right not sorted     i=1 means that the only sorted is in position 0.

#include <stdio.h>

void InsertionSort (int *A, int n);

int main(int argc, char *argv[]) {
  int n = 5;
  int arr[5] = {3, 8, 1, 7, 4};

  InsertionSort(arr, n);
  for(int i=0; i<n; i++){
    printf("%d ", arr[i]);
  }
  return 0;
}

void InsertionSort (int *A, int n) {
    int i, j, x;

    for (i=1; i<n; i++) {
        x = A[i];//first unsorted number
        j = i - 1;//J=0: only sorted number

        while (j>=0 && x<A[j]) {
            A[j+1] = A[j];//A[j] is not the smallest so it has to let the other go to left ex A[j+1=1]
            j--;//j=-1
        }

        A[j+1] = x;//A[j+1=0]
    }
}
```

1.2 Exchange Sort

O(n^2)
Vector divided in left unsorted and right sorted, swap of any numbers with a smaller one on its right

0	1	2	3	4	5
4	2	6	3	1	5
2	4	3	1	5	[6]
2	4	3	1	[5	6]
2	1	3	4	[5	6]
2	1	3	[4	5	6]
2	1	[3	4	5	6]
[1	2	3	4	5	6]

```
//2 subarray.
//left is unssorted, right is empty     i<n-1 means that you assume at the beginning the last is the great
void BubbleSort (int A[], int n) {
  int i, j, temp;

  for (i=0; i<n-1; i++) {
    for (j=0; j<n-i-1; j++){//j<n-1-i because the more you go on, less numbers you have left
      if (A[j] > A[j+1]) {
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
      }//in this for you find the greater unsorted number and swapping it, you put it on the right
  }
  }

  return;
}
```

1.3 Selection Sort

O(n^2)
Vector divided in left sorted and right unsorted, the algorithm looks for the smallest number in the right unsorted array and swaps it wiht the first element of the unsorted array

0	1	2	3	4	5
](4)	2	6	3	(1)	5
[1]	(2)	6	3	4	5
[1	2]	(6)	(3)	4	5
[1	2	3]	(6)	(4)	5
[1	2	3	4]	(6)	(5)
[1	2	3	4	5	6]

```
void SelectionSort (int A[], int n) {
    int i, j, min, temp;

    for (i=0; i<n-1; i++) {
        min = i;//first # is min

        for (j=i+1; j<n; j++) {   //it finds the smallest on the line
            if (A[j] < A[min]) {    //if the first on the right (A[j]) of the min num (A[min]) is less
                min = j;              //in that position there's the new smallest one
```

```
            }
        }                       //after this for you know which number is the smallest of the line
                                //end now you swap it with the one you assume to be the smallest (min=i

        temp = A[i];
        A[i] = A[min];
        A[min] = temp;
    }

  return;
}
```

1.4 Shell Sort

O(n^2)
swap numbers at same index in different arrays with insertion sort

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	h
7	6	8	9	8	6	2	1	8	7	0	4	5	3	0	1	0	4	9	
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
[3	0	1	0	4	6	2	1	5	7	3	4	5]	[7	6	8	9	8	9]	
[I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII]	[I	II	III	IV	V	VI]	13
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
[3	0	0	0]	[4	6	1	1]	[5	7	2	4]	[8	7	6	8]	[9	8	9]	
[I	II	III	IV]	[I	II	III	IV]	[I	II	III	IV]	[I	II	III	IV]	[I	II	III]	4
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0	0	0	1	4	6	1	1	5	7	2	4	8	7	6	8	9	8	9	1

```
void ShellSort (int A[], int n) {
    int i, j, x, h;
    h=1;

    while (h < n/3)
        h = 3*h+1;

    while (h >= 1) {
        for (i=h; i<n; i++) {
            x = A[i];
            j = i - h;

            while (j>=0 && x<A[j]) {
                A[j+h] = A[j];
                j -= h;
            }

            A[j+h] = x;
        }

        h = h/3;
    }
}
```

1.5 Counting Sort

O(n^2)
There are multiple arrays, the given one, another with every single value that is in the previous array with in each cell has the number of times that number exist in the previous array. A third array with

the cumulative number of element at each index. Another array with the previous array numbers shifted by 1 index to the right. At this point number 0 at index 0 is between position 0 and 1, 1 occurrence, number 1 at index 1 is between position 1 and 4, 3 occurrences in the last vectors, number 2 at index 2 is between position 4 and 4, 0 occurrences in the last vectors, and last number, 3 at index 3 is between position 4 and 6, 2 occurrences in the last array.

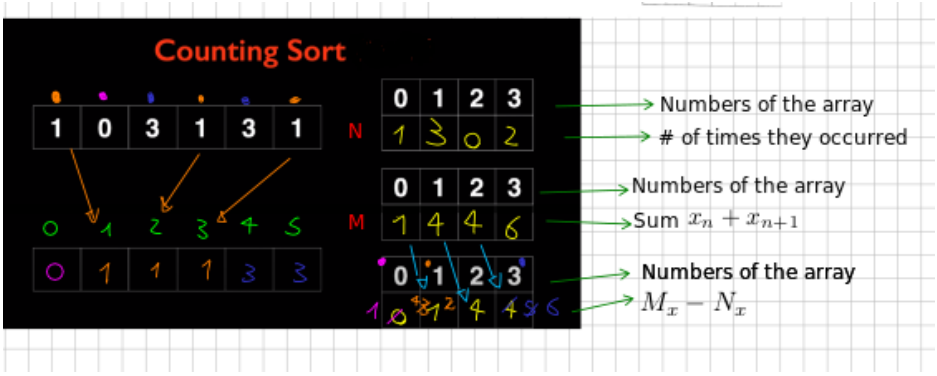

Figure 1: Counting sort

```c
#define MAX 100

void CountingSort(int A[], int n, int k) {
    int i, C[MAX], B[MAX];

    for (i=0; i<k; i++)
        C[i] = 0;

    for (i=0; i<n; i++)
        C[A[i]]++;

    for (i=1; i<k; i++)
        C[i] += C[i-1];

    for (i=n-1; i>=0; i--)
    {
        B[C[A[i]]-1] = A[i];
        C[A[i]]--;
    }

    for (i=0; i<n; i++)
        A[i] = B[i];
}
```

1.6 Merge Sort

O(log_n(n))
It divides the array and then, when it merges back the numbers, it does it ordering them

```c
#include <stdlib.h>
#include <stdio.h>

#define max 100

int insert_array(int V[]) {
  int n, i;
  printf("How many elements?: ");
```

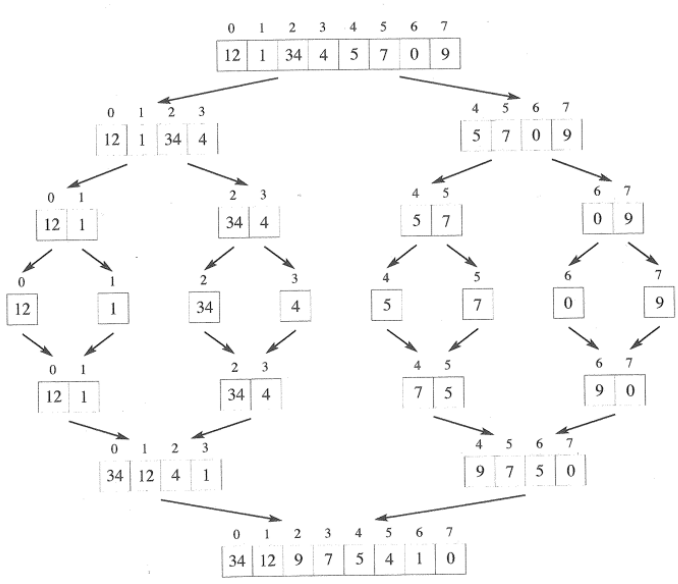

Figure 2: Merge sort

```
  scanf("%d", &n);

  for (i=0; i<n; i++) {
     printf("Element %d: ", i);
        scanf("%d", &V[i]);
  }
  return(n);
}

void print_array(int V[], int n) {
  int i;
  for (i=0; i<n; i++) {
    printf("%d ", V[i]);
  }
  printf("\n");
  return;
}

void merge(int a[], int p, int q, int r) {
  int i, j, k=0, b[max];
  i = p;
  j = q+1;

  while (i<=q && j<=r) {
    if (a[i]<a[j]) {
      b[k] = a[i];
      i++;
    } else {
      b[k] = a[j];
      j++;
    }
    k++;
  }
  while (i <= q) {
```

```
    b[k] = a[i];
    i++;
    k++;
  }
  while (j <= r) {
    b[k] = a[j];
    j++;
    k++;
  }
  for (k=p; k<=r; k++)
    a[k] = b[k-p];
  return;
}

void mergeSort(int a[], int p, int r) {
  int q;
  if (p < r) {
    q = (p+r)/2;
    mergeSort(a, p, q);
    mergeSort(a, q+1, r);
    merge(a, p, q, r);
  }
  return;
}

int main(void) {
  int n, V[max];
  n = insert_array(V);
  mergeSort(V, 0, n-1);
  print_array(V, n);
  return(0);
}
```

1.7 Quik Sort

O(log_n(n))
It divides the array and then, when it merges back the numbers, it does it ordering them

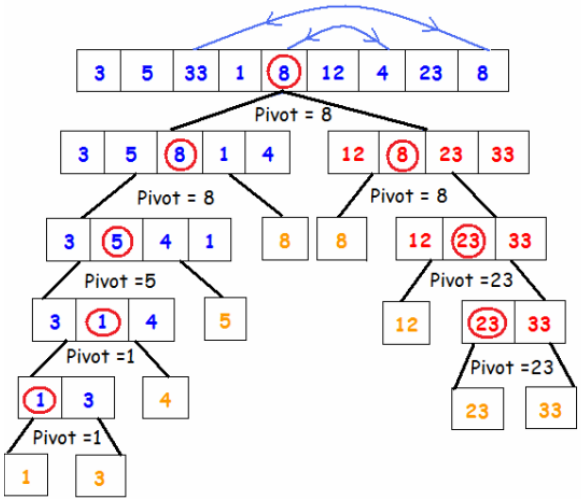

Figure 3: Quick sort

```c
#include<stdio.h>

void quicksort(int number[25],int first,int last){
  int i, j, pivot, temp;
  if(first<last){
    pivot=first;
    i=first;
    j=last;
    while(i<j){
      while(number[i]<=number[pivot]&&i<last)
        i++;
      while(number[j]>number[pivot])
        j--;
      if(i<j){
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }
    temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);
  }
}
int main(){
  int i, count, number[25];
  printf("Enter some elements (Max. - 25): ");
  scanf("%d",&count);
  printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);
  quicksort(number,0,count-1);
  printf("The Sorted Order is: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);
  return 0;
}
```

```
#ifndef _STACK_PUBLIC
#define _STACK_PUBLIC

#include <stdio.h>

/* macro definition */
#define stack_empty_m(sp) (stack_count(sp) == 0)

/* type declarations */
typedef struct stack stack_t;

/* extern function prototypes */
extern stack_t *stack_init(int size);
extern int stack_count(stack_t *sp);
extern int stack_push(stack_t *sp, void *data);
extern int stack_pop(stack_t *sp, void **data_ptr);
extern void stack_print(FILE *fp, stack_t *sp, void (*print)(FILE *, void *));
extern void stack_dispose(stack_t *sp, void (*quit)(void *));

#endif
```

```
#ifndef _STACK_PRIVATE
#define _STACK_PRIVATE

#include <stdio.h>
#include "stackPublic.h"
#include "util.h"

/* structure declarations */
struct stack {
  void **array;
  int index;
  int size;
};
```

```
#endi

#include "stackPrivate.h"

/*
 *  create a new empty stack
 */
stack_t *stack_init(int size) {
  stack_t *sp;

  sp = (stack_t *)util_malloc(sizeof(stack_t));
  sp->size = size;
  sp->index = 0;
  sp->array = (void **)util_malloc(size * sizeof(void *));
  return sp;
}

/*
 *  return the number of elements stored in the stack
 */
int stack_count(stack_t *sp) { return (sp != NULL) ? sp->index : 0; }

/*
 *  store a new value in the stack (LIFO policy)
 */
int stack_push(stack_t *sp, void *data) {
  if (sp == NULL || sp->index >= sp->size) {
    return 0;
  }

  sp->array[sp->index++] = data;
  return 1;
}

/*
 *  extract a value from the stack (LIFO policy)
 */
int stack_pop(stack_t *sp, void **data_ptr) {
  if (sp == NULL || sp->index <= 0) {
    return 0;
  }

  *data_ptr = sp->array[--sp->index];
  return 1;
}

/*
 *  print all the stack elements (LIFO policy)
 */
void stack_print(FILE *fp, stack_t *sp, void (*print)(FILE *, void *)) {
  int i;

  if (sp != NULL) {
    for (i = sp->index - 1; i >= 0; i--) {
      print(fp, sp->array[i]);
      fprintf(fp, "\n");
    }
```

```
  }
}

/*
 *  deallocate all the memory associated to the stack
 */
void stack_dispose(stack_t *sp, void (*quit)(void *)) {
  int i;

  if (sp != NULL) {
    if (quit != NULL) {
      for (i = 0; i < sp->index; i++) {
        quit(sp->array[i]);
      }
    }
    free(sp->array);
    free(sp);
  }
}
```

10.2 BST Library

```
#ifndef _TREE_PUBLIC_INCLUDED
#define _TREE_PUBLIC_INCLUDED

#include "data.h"

#define PREORDER  -1
#define INORDER    0
#define POSTORDER  1

typedef struct node node_t;

data_t getData (node_t *);
node_t *createEmptyTree ();
node_t *readTree(FILE *);
node_t *searchI (node_t *, data_t);
node_t *searchR (node_t *, data_t);
node_t *treeMinI (node_t *);
node_t *treeMinR (node_t *);
node_t *treeMaxI (node_t *);
node_t *treeMaxR (node_t *);
node_t *insert(node_t *, data_t);
node_t *delete(node_t *, data_t);
void writeTree(FILE *, node_t *, int);
void freeTree(node_t *);

#endif
```

```
#ifndef _TREE_PRIVATE_INCLUDED
#define _TREE_PRIVATE_INCLUDED

#include "treePublic.h"
// #include "treeAddition.h"

struct node {
  data_t val;
```

```
  struct node *left;
  struct node *right;
};


#endif
#include "treePrivate.h"

#define FIND 0

static node_t *myAlloc(void);
#if FIND
static data_t findDeleteMax1(node_t **);
#endif
#if !FIND
static node_t *findDeleteMax2(data_t *, node_t *);
#endif

data_t getData(node_t *node) { return (node->val); }

node_t *createEmptyTree(void) { return (NULL); }

node_t *treeMinI(node_t *rp) {
  if (rp == NULL)
    return (rp);

  while (rp->left != NULL) {
    rp = rp->left;
  }

  return (rp);
}

node_t *treeMinR(node_t *rp) {
  if (rp == NULL || rp->left == NULL)
    return (rp);

  return (treeMinR(rp->left));
}

node_t *treeMaxI(node_t *rp) {
  if (rp == NULL)
    return (rp);

  while (rp->right != NULL) {
    rp = rp->right;
  }

  return (rp);
}

node_t *treeMaxR(node_t *rp) {
  if (rp == NULL || rp->right == NULL)
    return (rp);

  return (treeMaxR(rp->right));
```

```
}

node_t *searchI(node_t *rp, data_t data) {
  while (rp != NULL) {
    if (compare(rp->val, data) == 0)
      return (rp);

    if (compare(data, rp->val) < 0)
      rp = rp->left;
    else
      rp = rp->right;
  }

  return (NULL);
}

node_t *searchR(node_t *rp, data_t data) {
  if (rp == NULL || compare(rp->val, data) == 0)
    return (rp);

  if (compare(data, rp->val) < 0)
    return (searchR(rp->left, data));
  else
    return (searchR(rp->right, data));
}

node_t *insert(node_t *rp, data_t data) {
  node_t *p;

  /* Empty Tree: Found Position */
  if (rp == NULL) {
    p = myAlloc();
    p->val = data;
    p->left = p->right = NULL;
    return (p);
  }

  /* Duplicated Element */
  if (compare(data, rp->val) == 0) {
    return (rp);
  }

  if (compare(data, rp->val) < 0) {
    /* Insert on the left */
    rp->left = insert(rp->left, data);
  } else {
    /* Insert on the right */
    rp->right = insert(rp->right, data);
  }

  return (rp);
}

node_t *readTree(FILE *fp) {
  node_t *rp;
  data_t d;
```

```
  rp = createEmptyTree();

  while (readData(fp, &d) != EOF) {
    rp = insert(rp, d);
  }

  return (rp);
}

void freeTree(node_t *rp) {
  if (rp == NULL) {
    return;
  }

  freeTree(rp->left);
  freeTree(rp->right);
  free(rp);

  return;
}

void writeTree(FILE *fp, node_t *rp, int modo) {
  if (rp == NULL) {
    return;
  }

  if (modo == PREORDER) {
    writeData(fp, rp->val);
  }

  writeTree(fp, rp->left, modo);

  if (modo == INORDER) {
    writeData(fp, rp->val);
  }

  writeTree(fp, rp->right, modo);

  if (modo == POSTORDER) {
    writeData(fp, rp->val);
  }

  return;
}

node_t *delete (node_t *rp, data_t data) {
  node_t *p;

  /* Empty Tree */
  if (rp == NULL) {
    printf("Error: Unknown Data\n");
    return (rp);
  }

  if (compare(data, rp->val) < 0) {
```

```
    /* Delete on the left sub-treee Recursively */
    rp->left = delete (rp->left, data);
    return (rp);
  }

  if (compare(data, rp->val) > 0) {
    /* Delete on the rigth sub-treee Recursively */
    rp->right = delete (rp->right, data);
    return (rp);
  }

  /* Delete Current Note rp */
  p = rp;
  if (rp->right == NULL) {
    /* Empty Right Sub-Tree: Return Left Sub-Tree */
    rp = rp->left;
    free(p);
    return (rp);
  }

  if (rp->left == NULL) {
    /* Empty Left Sub-Tree: Return Right Sub-Tree */
    rp = rp->right;
    free(p);
    return rp;
  }

  /* Find Predecessor and Substitute */
#if FIND
  rp->val = findDeleteMax1(&(rp->left));
#endif
#if !FIND
  {
    data_t val;

    rp->left = findDeleteMax2(&val, rp->left);
    rp->val = val;
  }
#endif

  return (rp);
}

static node_t *myAlloc(void) {
  node_t *p;

  p = (node_t *)malloc(sizeof(node_t));
  if (p == NULL) {
    printf("Allocation Error.\n");
    exit(1);
  }

  return (p);
}

#if FIND
```

```c
static data_t findDeleteMax1(node_t **rpp) {
  node_t *p;
  data_t d;

  /* Find The Rigth-Most Node (max value) */
  while ((*rpp)->right != NULL)
    rpp = &((*rpp)->right);

  p = *rpp;
  d = p->val;
  *rpp = (*rpp)->left;
  free(p);

  return (d);
}
#endif

#if !FIND
static node_t *findDeleteMax2(data_t *d, node_t *rp) {
  node_t *tmp;

  if (rp->right == NULL) {
    *d = rp->val;
    tmp = rp->left;
    free(rp);
    return (tmp);
  }

  rp->right = findDeleteMax2(d, rp->right);
  return (rp);
}
#endif
```

11 Item library

11.1 Item with Stack

```c
#ifndef _ITEM
#define _ITEM

#include <stdio.h>
#include "util.h"

/* type declarations */
typedef int *item_t;

/* extern function prototypes */
extern int item_read(FILE *fp, void **ptr);
extern void item_print(FILE *fp, void *ptr);
extern int item_compare(void *data1, void *data2);
extern void item_dispose(void *ptr);

#endif
```

```c
#include "item.h"

#define MAX 100
```

```c
/*
 *  read an item from file
 */
int item_read (FILE *fp, void **data_ptr) {
  int *p;

  p = (int *)util_malloc(sizeof(int));
  if (fscanf(fp, "%d", p) == EOF) {
    return EOF;
  }
  *data_ptr = p;

  return 1;
}

/*
 *  print an item on file
 */
void item_print (FILE *fp, void *ptr) {
  item_t data = (item_t)ptr;
  fprintf(fp, "%d ", *data);
}

/*
 *  compare two items
 */
int item_compare (void *ptr1, void *ptr2) {
  item_t data1 = (item_t)ptr1;
  item_t data2 = (item_t)ptr2;

  return (*data1)-(*data2);
}

/*
 *  free an item
 */
void item_dispose (void *ptr) {
  item_t data = (item_t)ptr;
  free(data);
  return;
}
```

11.2 Item with BST

12 Util library

12.1 Util with Stack

```c
#ifndef _UTIL
#define _UTIL

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* macro definition */
```

```
#define util_check_m(expr, msg) \
  if ( !(expr) ) { \
    fprintf(stderr, "Error: %s\n", msg); \
    exit(EXIT_FAILURE); \
  }

/* extern function prototypes */
extern FILE *util_fopen(char *name, char *mode);
extern void *util_malloc(unsigned int size);
extern void *util_calloc(unsigned int num, unsigned int size);
extern char *util_strdup(char *src);
extern void util_array_dispose(void **ptr, unsigned int n, void (*quit)(void *));
extern void **util_matrix_alloc(unsigned int n, unsigned int m, unsigned int size);
extern void util_matrix_dispose(void ***ptr, unsigned int n, unsigned int m,
                                void (*quit)(void *));
#endif
```

```
#include "util.h"

/*
 *  fopen (with check) utility function
 */
FILE *util_fopen(char *name, char *mode) {
  FILE *fp = fopen(name, mode);
  util_check_m(fp != NULL, "could not open file!");
  return fp;
}

/*
 *  malloc (with check) utility function
 */
void *util_malloc(unsigned int size) {
  void *ptr = malloc(size);
  util_check_m(ptr != NULL, "memory allocation failed!");
  return ptr;
}

/*
 *  calloc (with check) utility function
 */
void *util_calloc(unsigned int num, unsigned int size) {
  void *ptr = calloc(num, size);
  util_check_m(ptr != NULL, "memory allocation failed!");
  return ptr;
}

/*
 *  strdup (with check) utility function
 */
char *util_strdup(char *src) {
  char *dst = strdup(src);
  util_check_m(dst != NULL, "memory allocation failed");
  return dst;
}

/*
 *  array de-allocation utility function
```

```
 */
void util_array_dispose(void **ptr, unsigned int n, void (*quit)(void *)) {
  int i;

  if (quit != NULL) {
    for (i = 0; i < n; i++) {
      quit(ptr[i]);
    }
  }
  free(ptr);
}

/*
 *  matrix allocation utility function
 */
void **util_matrix_alloc(unsigned int n, unsigned int m, unsigned int size) {
  void **ptr;
  int i;

  ptr = (void **)util_malloc(n * sizeof(void *));
  for (i = 0; i < n; i++) {
    ptr[i] = util_calloc(m, size);
  }
  return ptr;
}

/*
 *  matrix de-allocation utility function
 */
void util_matrix_dispose(void ***ptr, unsigned int n, unsigned int m,
                         void (*quit)(void *)) {
  int i, j;

  for (i = 0; i < n; i++) {
    if (quit != NULL) {
      for (j = 0; j < m; j++) {
        quit(ptr[i][j]);
      }
    }
    free(ptr[i]);
  }
  free(ptr);
}
```

12.2 Util with BST

13 Data Library

13.1 Data with BST

```
#ifndef _DATA_INCLUDED
#define _DATA_INCLUDED

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXC 20

typedef int data_t;

int readData (FILE *, data_t *);
void writeData (FILE *, data_t);
int compare (data_t, data_t);

#endif
```

```
#include "data.h"

int readData(FILE *fp, data_t *data) {
  int retValue;

  retValue = fscanf(fp, "%d", data);

  return (retValue);
}

void writeData(FILE *fp, data_t data) {
  fprintf(fp, "%d\n", data);

  return;
}

int compare(data_t d1, data_t d2) {
  if (d1 < d2) {
    return (-1);
  } else {
    if (d1 == d2) {
      return (0);
    } else {
      return (1);
    }
  }
}
```

14 Symbol table

```
#ifndef ST_H_DEFINED
#define ST_H_DEFINED

#include "item.h"

typedef struct symboltable *ST;

ST      STinit(int) ;
void    STinsert(ST, Item) ;
Item    STsearch(ST, Key) ;
void    STdelete(ST, Key) ;
void    STdisplay(ST) ;

#endif
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "item.h"
#include "st.h"

typedef struct STnode *link;

struct STnode {
  Item item;
  link next;
};

struct symboltable {
  link *heads;
  int M;
  link z;
};

link NEW(Item item, link next) {
  link x = malloc(sizeof *x);
  x->item = item;
  x->next = next;
  return x;
}

ST STinit(int maxN) {
  int i;
  ST st = malloc(sizeof *st);

  st->M = maxN;
  st->heads = malloc(st->M * sizeof(link));
  st->z = NEW(ITEMsetvoid(), NULL);

  for (i = 0; i < st->M; i++)
    st->heads[i] = st->z;

  return st;
}

int hash(Key v, int M) {
  int h = 0, base = 127;

  for (; *v != '\0'; v++)
    h = (base * h + *v) % M;

  return h;
}

int hashU(Key v, int M) {
  int h, a = 31415, b = 27183;

  for (h = 0; *v != '\0'; v++, a = a * b % (M - 1))
    h = (a * h + *v) % M;
```

```
  return h;
}

void STinsert(ST st, Item item) {
  int i;

  i = hash(KEYget(item), st->M);

  fprintf(stdout, "  hash index = %d\n", i);

  st->heads[i] = NEW(item, st->heads[i]);

  return;
}

Item searchST(link t, Key k, link z) {
  if (t == z)
    return ITEMsetvoid();

  if ((KEYcompare(KEYget(t->item), k)) == 0)
    return t->item;

  return (searchST(t->next, k, z));
}

Item STsearch(ST st, Key k) {
  return searchST(st->heads[hash(k, st->M)], k, st->z);
}

link deleteR(link x, Key k) {
  if (x == NULL)
    return NULL;

  if ((KEYcompare(KEYget(x->item), k)) == 0) {
    link t = x->next;
    free(x);
    return t;
  }

  x->next = deleteR(x->next, k);

  return x;
}

void STdelete(ST st, Key k) {
  int i = hash(k, st->M);
  st->heads[i] = deleteR(st->heads[i], k);

  return;
}

void visitR(link h, link z) {
  if (h == z)
    return;

  ITEMshow(h->item);
```

```
  visitR(h->next, z);

  return;
}

void STdisplay(ST st) {
  int i;

  for (i = 0; i < st->M; i++) {
    fprintf(stdout, "st->heads[%d]: ", i);
    visitR(st->heads[i], st->z);
    fprintf(stdout, "\n");
  }

  return;
}
```

15 Item with Symbol tables

```
#ifndef _DATO_INCLUDED
#define _DATO_INCLUDED

#define MAXST 10

typedef struct item* Item;
typedef char *Key;

Item ITEMscan();
void ITEMshow(Item data);
int ITEMcheckvoid(Item data);
Item ITEMsetvoid();
Key KEYscan();
int KEYcompare(Key k1, Key k2);
Key KEYget(Item data);
#endif
```

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "item.h"

struct item {
  char *name;
  int value;
};

Item ITEMscan() {
  char name[MAXST];
  int value;

  scanf("%s%d", name, &value);

  Item tmp = (Item)malloc(sizeof(struct item));
  if (tmp == NULL) {
```

```
    return ITEMsetvoid();
  } else {
    tmp->name = strdup(name);
    tmp->value = value;
  }

  return tmp;
}

void ITEMshow(Item data) {
  fprintf(stdout, "   name = %s value = %d  ", data->name, data->value);
}

int ITEMcheckvoid(Item data) {
  Key k1, k2 = "";

  k1 = KEYget(data);
  if (KEYcompare(k1, k2) == 0)
    return 1;
  else
    return 0;
}

Item ITEMsetvoid() {
  char name[MAXST] = "";

  Item tmp = (Item)malloc(sizeof(struct item));
  if (tmp != NULL) {
    tmp->name = strdup(name);
    tmp->value = -1;
  }
  return tmp;
}
```