odmierzanie czasu - przykład

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/times.h>
#include <math.h>


  int main()
{int n;
  n=time(NULL);
  printf("\n  n=%d\n",n);
  printf("\n  ctime %s\n", ctime(  (const time_t *) &n ) );
} /* koniec main */
```

odmierzanie czasu - przykład

n=1511227738

ctime Tue Nov 21 02:28:58 2017

errno – **kod błędu**

Większość funkcji bibliotecznych zwraca szczególną wartość dla zaznaczenia, że zawiodły. np. –1, zerowy wskaźnik NULL, stałą jak EOF itp.. Mówi to jednak tylko, że zdażył się błąd. Jaki? należy skontrolować jaki **kod błędu** został zapisany w errno. Zmienna erno jest zadeklarowana w errno.h . Kody błędu są przedstawione w opisach funkcji.

Funkcje nie zmieniają wartości errno jeśli błędu nie było. Tym samym wartość errno po udanym zawołaniu funkcji niekoniecznie jest zero!

Oczywiście po tym jak funkcja zakończyła się błędnie, można sprawdzić, co zapisane jest w errno; można także wyzerować errno przed zawołaniem funkcji.

errno

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int n;
    main()
{                   /*     rmdir("testowa_kartoteka");   */
     n=mkdir("testowa_kartoteka",0774);
     if(errno==EEXIST) printf("\n  !! EEXIST=%d\n",EEXIST);
   printf("\n n=%d\n",n);
   exit(0);  }                    /* koniec   funkcji  main*/
```

errno

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
   int main ()
{  int k;
    k=unlink("out");
    if(k == -1)        perror ("unlink!!");
    exit(0);
}       /*   koniec funkcji    main  */
```

errno

void perror (const char *s);

funkcja perror wypisuje komunikat na pliku stderr. Najpierw wypisany zostaje łańcuch znakowy s , następnie dwukropek i spacja. Następnie wypisywany jest komunikat, jest on zakończony znakiem \n (czyli znakiem nowej linii).

funkcja **sync**

W nowoczesnych systemach operacyjnych operacje WE/WY nie są wykonywane synchronicznie, tzn. nawet jeśli np. write się skończyło, nie znaczy to,że dane zostały fizycznie zapisane na dysku. jeśli synchronizacja jest konieczna, można użyc specjalnych funkcji które powodują, że dane zostają zapisane np.. na dysku.

#include <unistd.h>

int **sync** *(void)*

Po wywołaniu funkcji sterowanie nie zostanie zwrócone tak długo, jak długo dane nie zostaną zapisane na urządzeniu.

Funkcja zwraca zero, jeśli nie było błędu.

Oczywiście, nie wiadomo ile czasu sync będzie musiała zużyć na synchronizację danych.

funkcja **fsync**

#include <unistd.h>

int **fsync** *(int deskryptor_pliku)*

Użycie funkcji fsync powoduje, że wszystkie dane związane z otwartym plikiem o deskryptorze *deskryptor_pliku* zastają zapisane na urządzeniu. Funkcja nie zwraca sterowania do miejsca skąd została wywołana, dopóki nie zakończy pozytywnie synchronizacji lub dopóki nie nastąpi błąd.Funkcja zwraca zero jeśli nie było błędu, -1 jeśli był błąd. Ustawia zmienną errno w przypadku błędu.

EBADF

deskryptor pliku jest niepoprawny

EINVAL

synchronizacja nie jest możliwa gdyż system tego nie przewiduje

funkcje **fstat** i **stat**

Te dwie funkcje umożliwiają uzyskanie informacji o pliku. Można np. odczytać jak wielki jest plik czy kiedy został utworzony. Prototypy tych funkcji są w **<sys/stat.h>**

int **stat**(char *path, struct stat *buf),

int fstat(int fd, struct stat *buf)

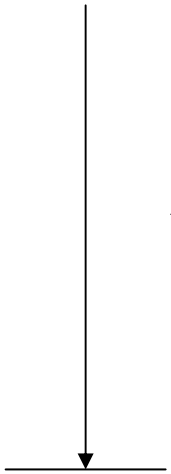
stat(..) uzyskuje informacje o pliku którego nazwa jest wskazana przez path.

fstat(..) czyni to samo, jako pierwszego argumentu używa deskryptora pliku taki jaki jest otrzymywany przy użyciu funkcji open (open otwiera plik do operacji niebuforowanych na nim)

Obie funkcje zwracają wartość 0 w przypadku sukcesu, -1 w przypadku błędu (ustawiają także kody błędu w errno)

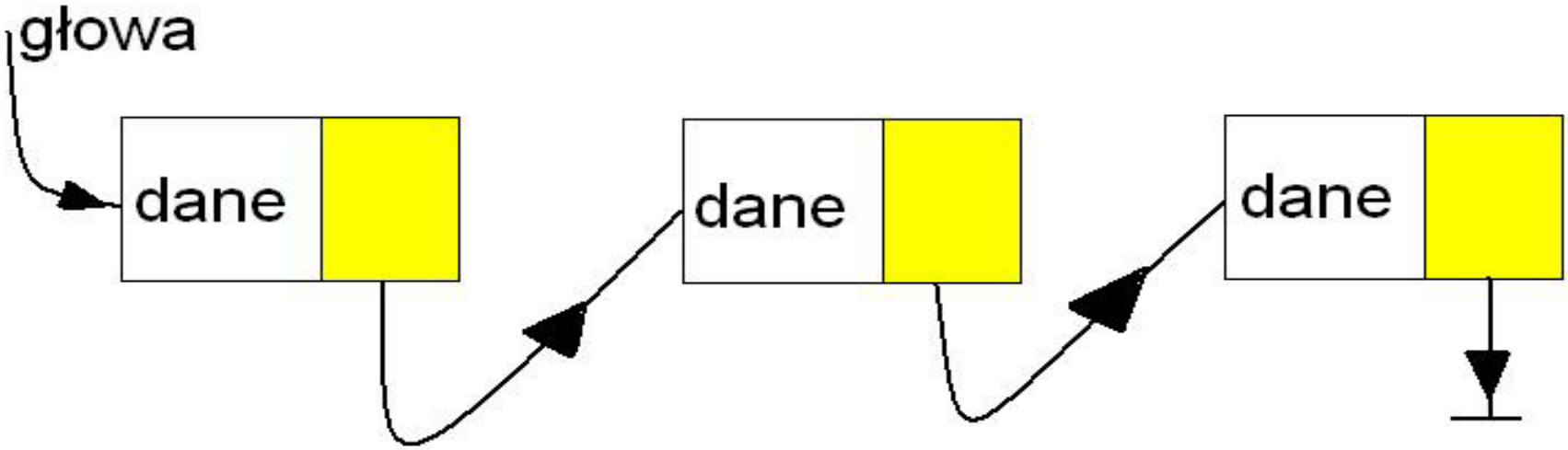
funkcje **fstat** i **stat**

```
struct stat {
off_t    st_size; /* rozmiar pliku w bajtach pliku*/
time_t st_atime; /* czas ostatniego dostępu pliku*/
time_t st_mtime; /* czas ostatniej modyfikacji pliku*/
time_t st_ctime; /* czas ostatniej zmiany statusu pliku */
/* czas mierzony w sekundach od
        00:01:00 UTC, Jan. 1, 1970 */
blkcnt_t st_blocks; /* liczba zaalokowanych bloków po 512 b*/
}
```



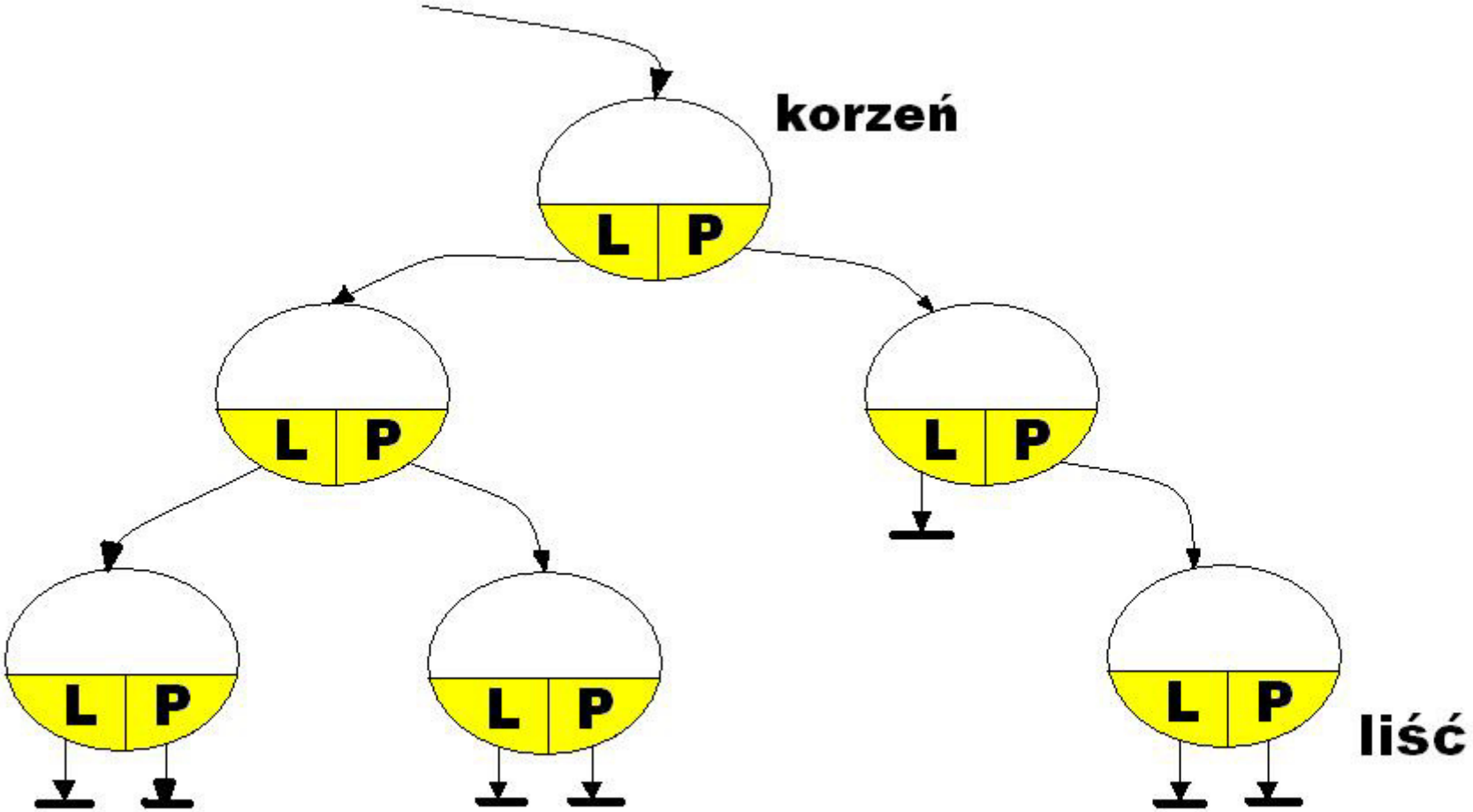
wskaźnik zerowy NULL

NULL



lista jednokierunkowa

struktury danych
drzewo



węzeł główny, węzeł, rodzic, dziecko, liść, współczynnik rozgałęzienia

struktury danych-lista1

```
/*                                                    */

/* Przyklad: dynamiczne struktury danych w  C          */


#include <stdio.h>


#define FALSE 0


typedef struct {
   int     dataitem;
   struct listelement *link;
}              listelement;
```

struktury danych-lista2

```
void Menu (int *choice);

listelement * Add (listelement * listpointer, int data);

listelement * Remove (listelement * listpointer);

void PrintQueue (listelement * listpointer);

void ClearQueue (listelement * listpointer);
```

```
 int   main ()
{   /*  początek  funkcji  main  */
   listelement listmember, *listpointer;
   int     data,
           choice;

   listpointer = NULL;
```

```
do {
        Menu (&choice);
        switch (choice) {
            case 1:
                printf ("Enter data item value to add  ");
                scanf ("%d", &data);
                listpointer = Add (listpointer, data);
                break ;
```

```
case 2:
        if (listpointer == NULL)
            printf ("Queue empty!\n");
        else
            listpointer = Remove (listpointer);
        break;
```

```
case 3:
        PrintQueue (listpointer);
        break;
case 4:
        break;
default:
        printf ("Invalid menu choice - try again\n");
        break;
    }
  } while (choice != 4);
```

```
 ClearQueue (listpointer);
}                              /*   koniec  funkcji    main */
```

```
void Menu (int *choice) {
    char    local;
    printf ("\nEnter\t1 to add item,\n\t2 to remove item\n\
\t3 to print queue\n\t4 to quit\n");
    do {
        local = getchar ();
        if ((isdigit (local) == FALSE) && (local != '\n'))
        {  printf ("\nyou must enter an integer.\n");
           printf ("Enter 1 to add, 2 to remove, 3 to print, 4 to quit\n");
        }
    } while (isdigit ((unsigned char) local) == FALSE);
    *choice = (int) local - '0';                    } /*   koniec  funkcji   Menu   */
```

```
listelement * Add (listelement * listpointer, int data) {
    listelement * lp = listpointer;
    if (listpointer != NULL) {
        while (listpointer -> link != NULL)
            listpointer = (listelement *) listpointer -> link;
        listpointer -> link = (struct listelement  *) malloc
                (sizeof (listelement));
        listpointer = (listelement *) listpointer -> link;
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return lp;
    }
```

```
else {
        listpointer = (listelement  *) malloc (sizeof (listelement));
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return listpointer;
    }
}   /*     koniec      funkcji      Add   */
```

```
listelement * Remove (listelement * listpointer) {


    listelement * tempp;
    printf ("Element removed is %d\n", listpointer -> dataitem);
    tempp = (listelement *) listpointer -> link;
    free (listpointer);
    return tempp;
}   /*   koniec  funkcji   Remove    */
```

```
void PrintQueue (listelement * listpointer) {


   if (listpointer == NULL)
      printf ("queue is empty!\n");
   else
      while (listpointer != NULL) {
         printf ("%d\t", listpointer -> dataitem);
         listpointer = (listelement *) listpointer -> link;
      }
   printf ("\n");     }   /*    koniec  funkcji   PrintQueue  */
```

```
void ClearQueue (listelement * listpointer) {


    while (listpointer != NULL) {
        listpointer = Remove (listpointer);
    }   /*   koniec   pętli  while   */
}     /*  koniec    funkcji    ClearQueue   */
```

cechy struktur danych

Mówiąc o strukturach danych ma się na myśli określone działania, czyli operacje, które można na nich wykonywać; przykładowo dla listy są to operacje wstawiania, usuwania, przeglądania i zliczania elementów listy. Struktura danych wraz z podstawowymi na niej operacjami nazywana jest ***abstrakcyjnym typem danych***

(listy, stosy, kolejki, drzewa, sterty...)

Algorytmy

- wykorzystanie giętkości języka programowania

-algorytmy pozwalają oderwać się od kodu, gdyż wiele złożonych problemów można podzielic na mniejsze części, dla których już istnieją przetestowane algorytmy postępowania (sortowanie, kodowanie,wyszukiwanie...)

-błędnie uważa się, że postępowanie skomplikowane jest bardzo mądre; tymczasem najlepsze algorytmy zwykle są najprostsze. Niestety zwykle te najprostsze rozwiązania, algorytmy najtrudniej jest znaleźć...(na szczęście jest literatura)

Algorytmy

Klasyfikacja algorytmów – na osobnym zbiorze

algorytmy.klasyfikacja.rtf

Sortowanie pęcherzykowe czyli ”bubble sort”

przykład: users.uj.edu.pl/~ufrudy/bubble1.c

zdanie niektórych osób:

”...if you know what bubble sort is, wipe it from your mind; if you do not know, make a point of never finding out!...”

”bubble sort”

```
void bubbleSort(int numbers[], int array_size)
{ int i, j, temp;
  for (i = (array_size - 1); i >= 0; i--)
  {     for (j = 1; j <= i; j++)
        { if (numbers[j-1] > numbers[j])
          {     temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
           }
    }
  }
} /*  koniec  funkcji bubbleSort      */
```

Sortowanie przez wstawienie

Sortowanie przez wstawianie jest jednym z najprostszych algorytmów sortowania. Nieodpowiedni w przypadku dużych zbiorów, gdyż określenie miejsca kolejnych elementów w zbiorze może wymagać wielokrotnego przejrzenia wszystkich elementów tam umieszczonych.

Ale: ważną zaletą sortowania przez wstawianie jest to, że wstawienie pojedynczego elementu do już posortowanego pliku wymaga tylko jednokrotnego przejrzenia elementów, a nie powtórzenia całego algorytmu. Z tego powodu sortowanie przez wstawianie jest dobrym rozwiązaniem w przypadku sortowania przyrostowego.

Przykład: rezerwacja miejsc w dużym hotelu. Przy zastosowaniu sortowania przez wstawianie wystarczy jeden raz przejrzeć listę.

Sortowanie przez wstawienie (program)

```
#include <stdlib.h>

#include <string.h>


int issort(void *data, int size, int esize, int (*compare)(const void
*key1,  const void *key2));

/* wskaźnik do danych, ilość danych, rozmiar pojedynczej danej,

nazwa funkcji która będzie porównywała pojedyncze dane */

int compare (const void * x, const void * y);
```

Sortowanie przez wstawienie (program)

```
int issort(void *data, int size, int esize, int (*compare)(const void
*key1,  const void *key2))

{

char              *a = data;

void              *key;

int               i,  j;


/*  Alokacja pamięci na element key.        */

if ((key = (char *)malloc(esize)) == NULL)

   return  -1;
```

Sortowanie przez wstawienie (program)

```
/*  Kolejne wstawianie elementu key między elementy posortowane.*/
for (j = 1; j < size; j++)
{ memcpy(key, &a[j * esize], esize);
  i = j - 1;
/*  Określenie położenia, gdzie należy wstawić element key. */
  while (i >= 0 && compare(&a[i * esize], key) > 0)
  {  memcpy(&a[(i + 1) * esize], &a[i * esize], esize);
     i--;
  }
  memcpy(&a[(i + 1) * esize], key, esize);
}
```

Sortowanie przez wstawienie (program)

```
/* Zwolnienie pamięci zaalokowanej na potrzeby sortowania.    */
free(key);
return 0;
}  /*  koniec  funkcji issort            */
```

Sortowanie przez wstawienie (program)

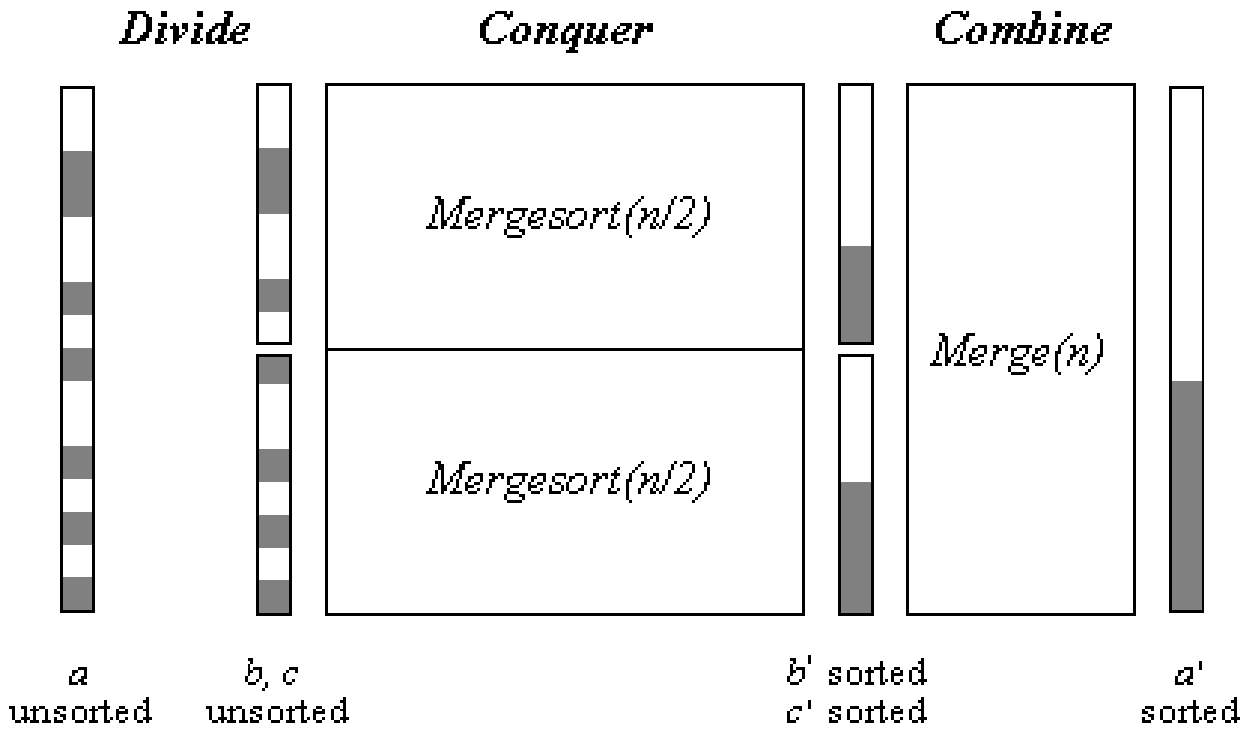
```
int compare (const void * x, const void * y)
{
  int *a, *b;
  a=(int *) x;
  b=(int *) y;
  if( (*a) > (*b) ) return(1);
  else
  return(-1);
} /*  koniec  funkcji  compare  */
```

Sortowanie przez wstawienie (program)

```
int  main ()
{
   int n;
   int bak[6]={3,5,1,2,4,0};
        for(n=0;n<6;++n) printf(" %d ", *(bak+n) );
        printf("\n");

            issort( bak, 6,4,compare);
       for(n=0;n<6;++n) printf(" %d ", *(bak+n) );
       printf("\n");
}  /*  koniec funkcji main  */
```

Dziel – Rządź - Złącz



sort złączenia (”Merge Sort”) cz. 1

```
#include <stdlib.h>

#include <string.h>


int mgsort(void *data, int size, int esize, int i, int k, int
(*compare)  (const void *key1, const void *key2));


static int merge(void *data, int esize, int i, int j, int k,
int (*compare)  (const void *key1, const void *key2))

{
```

sort złączenia (”Merge Sort”) cz. 2

```
char            *a = data, *m;
int             ipos,   jpos,  mpos;
/* inicjalizacja liczników złączania  */
ipos = i;
jpos = j + 1;
mpos = 0;
        /*  Rezerwacja pamięci na złączane elementy  */
```

sort złączenia (”Merge Sort”) cz. 3

```
if ((m = (char *) malloc(esize * ((k - i) + 1))) == NULL)
   return -1;
/*  Działamy, póki którakolwiek część ma elementy do złączania  */
while (ipos <= j || jpos <= k) {
   if (ipos > j) {/*  W lewej części nie ma już elementów do
złączania */
 while (jpos <= k) {
        memcpy(&m[mpos * esize], &a[jpos * esize], esize);
        jpos++;
        mpos++;
       }
```

sort złączenia (”Merge Sort”) cz. 4

```
 continue;
      }
   else if (jpos > k) {/*  w prawej części nie ma już elementów do
złączania  */
 while (ipos <= j) {
        memcpy(&m[mpos * esize], &a[ipos * esize], esize);
        ipos++;
        mpos++;
      }
     continue;
   }
```

sort złączenia (”Merge Sort”) cz. 5

```
/*  Dołączenie do złączonych elementów następnego elementu  */
if (compare(&a[ipos * esize], &a[jpos * esize]) < 0) {

    memcpy(&m[mpos * esize], &a[ipos * esize], esize);
    ipos++;
    mpos++;

    }
```

sort złączenia (”Merge Sort”) cz. 6

```
else {
      memcpy(&m[mpos * esize], &a[jpos * esize], esize);
      jpos++;
      mpos++;
   }
}
```

sort złączenia (”Merge Sort”) cz. 7

```
/*  Przygotowanie do przekazania z powrotem złączonych danych  */
memcpy(&a[i * esize], m, esize * ((k - i) + 1));

/*  Zwolnienie pamięci używanej na złączanie.                  */
free(m);
return 0;
}  /*  koniec  funkcji   merge   */
```

sort złączenia (”Merge Sort”) cz. 8

```
int mgsort(void *data, int size, int esize, int i, int k, int (*compare)
   (const void *key1, const void *key2)) {
        int    j;
/*  Koniec rekurencji, kiedy niemożliwe są dalsze podziały  */
if (i < k) {
/*  Ustalenie, gdzie należy podzielić elementy.                */
     j = (int)(((i + k - 1)) / 2);
```

sort złączenia (”Merge Sort”) cz. 9

```
 /*  Rekurencyjne sortowanie dwóch części  */

 if (mgsort(data, size, esize, i, j, compare) < 0)    return -1;

 if (mgsort(data, size, esize, j + 1, k, compare) < 0)   return -1;

 /*  Złączanie dwóch posortowanych części w jeden zbiór
posortowany  */

    if (merge(data, esize, i, j, k, compare) < 0)  return -1;

}

return 0;

}
```

sort złączenia (”Merge Sort”) cz. 10

```
int compare (const void * x, const void * y)
{
  int *a, *b;
  a=(int *) x;
  b=(int *) y;
  if( (*a) > (*b) ) return(1);
  else
  return(-1);
} /*  koniec  funkcji  compare  */
```

sort złączenia (”Merge Sort”) cz. 11

```
/* program główny testuje  funkcję  mgsort  */
int  main()
{ int a[5];
   int n;
   int ktrl=-99;;
         srand(getpid());
  for(n=0;n<5;++n)
  {  a[n]=rand()%100;
     printf("  %d", *(a+n));
  }
```

sort złączenia (”Merge Sort”) cz. 12

```
ktrl=mgsort(a, 5, sizeof(int), 0,4,compare);  /* sortowanie !!  */
 printf("\n   ktrl=%d",ktrl);
 printf("\n\n");
 for(n=0;n<5;++n)
 {  printf("  %d", *(a+n)); /* to elementy posortowane   */
 }  exit(0);
}/*  koniec  funkcji  main  */
```

sort złączenia (”Merge Sort”) cz. 13

users.uj.edu.pl/~ufrudy/mgsorttest.c

algorytm sortowania QUICKSORT

- algorytm Mergesort ma pewną wadę.....

- statystycznie dobrany ”punkt podziału” - QUICKSORT

- wady QUICKSORT; co jeśli ”punkt podziału” jest dobrany niefortunnie ?

QUICKSORT-1

```
#include <stdlib.h>

#include <string.h>


int issort(void *data, int size, int esize, int
(*compare)(const void *key1,

   const void *key2));/* issort to sort przez wstawienie  */

int compare (const void * x, const void * y);
```

QUICKSORT-2

```
static int compare(const void *int1, const void *int2) {
/* Porównanie dwóch liczb całkowitych  */

if (*(const int *)int1 > *(const int *)int2)
   return 1;
else if (*(const int *)int1 < *(const int *)int2)
   return -1;
else
   return 0;
}/*  koniec funkcji  compare   */
```

QUICKSORT-3

```
static int partition(void *data, int esize, int i, int k, int
(*compare) (const void *key1, const void *key2)) {

char            *a = data;

void            *pval,  *temp;

int             r[3];

/* rezerwacja pamieci na wartosc podzialu

i zamiane wartosci miejscami  */
```

QUICKSORT-4

```
if ((pval = malloc(esize)) == NULL)
   return -1;
if ((temp = malloc(esize)) == NULL) {
  free(pval);
   return -1;
}
```

QUICKSORT-5

```
/* Znalezienie wartosci podziału metodą  potrójnej
mediany  */

   r[0] = (rand() % (k - i + 1)) + i;

   r[1] = (rand() % (k - i + 1)) + i;

   r[2] = (rand() % (k - i + 1)) + i;

issort(r, 3, sizeof(int), compare);  /* sortowanie przez
wstawienie    */

memcpy(pval, &a[r[1] * esize], esize);
```

QUICKSORT-6

```
/* Utworzenie dwóch części  wokół wartości podziału */

i--;

k++;

while (1) {

/*  Przechodzimy w lewo, aż znajdziemy element
znajdujący się w niewłaściwej części   */

do {

     k--;

   } while (compare(&a[k * esize], pval) > 0);
```

QUICKSORT-7

```
/* Przechodzimy w prawo, aż znajdziemy element
znajdujący się  w niewłaściwej części  */

do {

     i++;

   } while (compare(&a[i * esize], pval) < 0);

if (i >= k) {

   */  Koniec dzielenia, kiedy zetkną się liczniki lewy i
         prawy.          */

     break;

              }
```

QUICKSORT-8

```
else {    /* Zamiana miejscami elementów z lewego i
                  prawego licznika          */


     memcpy(temp, &a[i * esize], esize);

     memcpy(&a[i * esize], &a[k * esize], esize);

     memcpy(&a[k * esize], temp, esize);

      }
}
```

QUICKSORT-9

```
/*  Zwolnienie pamięci zarezerwowanej na podział  */

free(pval);

free(temp);


/*  Zwrócenie położenia dzielącego części   */

return k;

}   /*    koniec  funkcji  partition    */
```

QUICKSORT-10

```
int quicksort(void *data, int size, int esize, int i, int k, int
(*compare) (const void *key1, const void *key2))

{

   int              j;

  /* Jeśli niemożliwe dalsze podziały, koniec rekurencji */


if (i < k)  {
```

QUICKSORT-11

```
/*  Sprawdzenie, gdzie należy dokonać podziału  */


   if ((j = partition(data, esize, i, k, compare)) < 0)
      return -1;
```

QUICKSORT-12

```
/*  Rekurencyjne sortowanie lewej części  */
if (quicksort(data, size, esize, i, j, compare) < 0)
      return -1;
/*  Rekurencyjne sortowanie prawej części   */
   if (quicksort(data, size, esize, j + 1, k, compare) < 0)
      return -1;
}
return 0;
}/* koniec funkcji  quicksort    */
```

złożoność algorytmu (jak rośnie czas)

sortowanie bąbelkowe **O(n*n)**

sortowanie przez wstawienie **O(n*n)**

sort złączenia **O(n*log(n))**

quicksort **O(n*log(n))**

sortowanie na stercie **O(n*log(n))**

sortowanie ze zliczaniem **O(n+k)** (k-to największa liczba powiększona o jeden)

sortowanie na bazie **O(pn + pk)** (k to baza, p to liczba cyfr w liczbach)

sortowanie ze zliczaniem

szybki algorytm sortowania, który opiera się na zliczaniu wystąpień poszczególnych elementów w zbiorze.

niestety, ma ograniczenia dotyczące danych

a)sortować można jedynie liczby całkowite i dane dające się jako takie wyrazić

b)trzeba znać największą liczbę występująca w danych (rezerwuje się odpowiednio dużą tablicę)

sortowanie ze zliczaniem - 1

```
#include <stdlib.h>
#include <string.h>
int ctsort(int *data, int size, int k) {
 /*  size –ile elementów, k –największa wartość  */
int             *counts,
                *temp;
int             i,  j;
```

sortowanie ze zliczaniem - 2

```
/*  Alokacja pamięci na liczniki.  */
if ((counts = (int *) malloc(k * sizeof(int))) == NULL)
    return -1;

/*  Alokacja pamięci na posortowane elementy.  */
if ((temp = (int *)malloc(size * sizeof(int))) == NULL)
   return -1;
```

sortowanie ze zliczaniem - 3

```
/*  Inicjalizacja liczników.  */
   for (i = 0; i < k; i++)
   counts[i] = 0;

/*  Zliczanie wystąpień poszczególnych elementów.  */
   for (j = 0; j < size; j++)
   counts[data[j]] = counts[data[j]] + 1;
/*   ++counts[ *(data+j) ];  */
/*       ++(*( counts + (*(data+j)) )) ;  */
```

sortowanie ze zliczaniem - 4

```
/*  Korekta poszczególnych liczników, aby zawierały liczbę
poprzedzających  je elementów  */

    for (i = 1; i < k; i++)

    counts[i] = counts[i] + counts[i - 1];

/*  Użycie counts do położenia poszczególnych elementów w
odpowiednie miejsca.*/

   for (j = size - 1; j >= 0; j--)

{ temp[counts[data[j]] - 1] = data[j];  /* tu temp[] użyte  */

  counts[data[j]] = counts[data[j]] - 1;

}
```

sortowanie ze zliczaniem - 5

```
/*  Przygotowanie posortowanych danych do zwrócenia*/
        memcpy(data, temp, size * sizeof(int));/*
/*Zwolnienie pamięci użytej do sortowania  */

        free(counts);

        free(temp);

return 0;

}  /*  koniec   funkcji  ctsort  */
```

sortowanie ze zliczaniem - 6

```
int  main()
{int a[5];
  int n;
  int ktrl=-99;;
  srand(getpid());
for(n=0;n<5;++n)
  {
    a[n]=rand()%100;
    printf("  %d", *(a+n)); /* wartości niesortowane  */
  }
```

sortowanie ze zliczaniem - 7

```
ktrl=ctsort(a, 5, 100);  /* wywołanie f. sortującej  */
  printf("\n   ktrl=%d",ktrl);
  printf("\n\n");
  for(n=0;n<5;++n)
  {
    printf("  %d", *(a+n));  /* po przesortowaniu  */
  }
}/*  koniec  funkcji  main  */
```

sortowanie ze zliczaniem - 8

sortowanie ze zliczaniem to szybki algorytm sortowania czasu liniowego (chodzi o to, że jego tzw. „złożoność” jest O(n+k), gdzie n jest liczbą sortowanych liczb całkowitych, a k jest największą z tych liczb powiększoną o jeden

sortowanie ze zliczaniem - 9

Adres strony:

users.uj.edu.pl/~ufrudy/ctsorttest.c

Sortowanie na bazie

polega na sortowaniu fragmentów danych, *ciągów cyfr*, od cyfry najmniej znaczącej do najbardziej znaczącej; jeśli np. sortujemy zbiór liczb zapisanych w systemie dziesiętnym

[15,12,49,16,36,40]

po posortowaniu cyfr najmniej znaczących otrzymuje się

[40,12,15,16,36,49]

po posortowaniu cyfr bardziej znaczących

[12,15,16,36,40,49]

Sortowanie na bazie

w sortowaniu na bazie korzysta się z sortowania ze zliczeniem, dzięki czemu algorytm jest szybki

Sortowanie na stercie
Heap Sort

Zbiór n liczb tworzy stertę (”heap”), jeżeli jego elementy spełniają następujący związek:

$a_{j/2}$

$a_{j/2} \geq a_j$ dla $1 \leq j/2 < j \leq n$

(j/2 oznacza dzielenie całkowitoprzecinkowe)

Sortowanie na stercie
Heap Sort

Kopiowanie na stercie - algorytm ten najpierw buduje stertę a następnie „zdejmuje” element ze szczytu sterty, powoduje to szereg awansów, następnie znowu „zdejmuje” element ze szczytu sterty, i tak dalej aż wszystkie elementy zostaną „zdjęte” czyli posortowane