Structuri de date și algoritmi

Universitatea Politehnica Timişoara
P-ța Victoriei nr. 2
RO 300006 - Timişoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 5

Ș.l. dr.ing. Adriana ALBU
adriana.albu@upt.ro
http://www.aut.upt.ro/~adrianaa

Universitatea
Politehnica
Timişoara

P-ţa Victoriei nr. 2
RO 300006 - Timişoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

3. Tehnici de sortare (partea a doua)

3.5 Tehnica sortării prin diminuarea incrementului – Shell

- Generalizează tehnica sortării prin inserție
- Recapitulare – tehnica sortării prin inserție:
- Sortarea prin inserție
  - nu e performantă când elemente apropiate ca valoare sunt depărtate ca poziționare în tablou
- Shell Sort
  - reduce distanța între elemente de valori apropiate
  - execută un număr mai mic de mutări

d a m x b d n

a d m x b d n

a d m x b d n

a d m x b d n

a b d m x d n

a b d d m x n

a b d d m n x

3.5 Tehnica sortării prin diminuarea incrementului – Shell

- **Principiul** metodei de sortare prin inserție cu diminuarea incrementului:
  - se realizează **sortări prin inserție repetate** asupra elementelor tabloului
  - se folosește un **„pas de sortare” – incrementul** secvenței supusă sortării
    - un **sir descrescător** de valori
- „Pasul de sortare”
  - precizează **distanța** între elementele care alcătuiesc secvența supusă sortării
  - este posibilă orice secvență descrescătoare de incremenți
    - cu condiția ca **ultimul increment să fie 1**
  - notații: $h_1, h_2, \ldots, h_t$; $h_t=1$; $h_k > h_{k+1}$; (exemplu secvență incremenți: 3, 2, 1)
  - Nu este indicat ca incremenții să fie puteri ale aceleiași baze, pentru a nu se prelucra aceleași elemente de mai multe ori (4, 2, 1)
- Fiecare sortare succesivă profită de sortările anterioare

3.5 Tehnica sortării prin diminuarea incrementului – Shell

- Fiecare *sortare-h* implică
  - relativ puține elemente
  - elemente care sunt deja sortate
- Nu este o sortare stabilă – poate modifica ordinea elementelor cu valori egale
- Garantează că o secvență *sortată-k*, rămâne *sortată-k* și după o *sortare-l*
- O eficiență sporită pentru sortarea shell se obține dacă între diferitele secvențe de parcurgere au loc cât mai puține interacțiuni
- Sunt recomandate diferite secvențe de incremenți:
  - Knuth
    - $h_{k-1} = 3h_k + 1$; $h_t = 1$; $t = \log_3 n - 1$
    - $h_t=1$, $h_{t-1}=4$, $h_{t-2}=13$, …
  - Hibbard
    - $h_{k-1} = 2h_k + 1$; $h_t = 1$; $t = \log_2 n -1$
    - 1, 3, 7, 15,…, 2^k-1

$O(n^{5/4})$

3.5 Tehnica sortării prin diminuarea incrementului – Shell

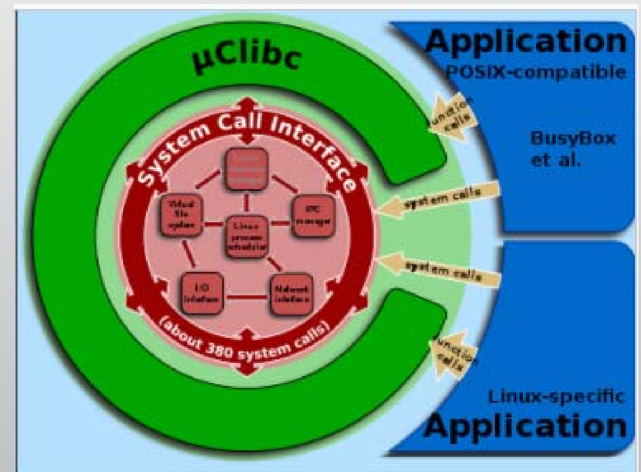
```
void shellSort(char *s, int n){
int i, j, k, w;
char x;
int h[3]={3,2,1};
      for(w=0;w<3;w++){
            k=h[w];
            for(i=k;i<n;++i){
                  x=s[i];
                  j=i-k;
                  while(x<s[j]&&j>=0){
                        s[j+k]=s[j];
                        j=j-k;
                  }
                  s[j+k]=x;
            }
      }
}
```

Exemplu: tablou: d, a, m, y, b, d, n, p, c, t
incremenți: 3, 2, 1

Trecere 1, Pas 3 3 sub-tablouri	d	a	m	y	b	d	n	p	c	t
	d	a	d	y	b	m	n	p	c	t
	d	a	d	n	b	m	y	p	c	t
	d	a	c	n	b	d	y	p	m	t
	d	a	c	n	b	d	t	p	m	y
Trecere 2, Pas 2 2 sub-tablouri	d	a	c	n	b	d	t	p	m	y
	c	a	d	n	b	d	t	p	m	y
	b	a	c	n	d	d	t	p	m	y
	b	a	c	d	d	n	t	p	m	y
	b	a	c	d	d	n	m	p	t	y
Tr. 3, Pas 1	b	a	c	d	d	n	m	p	t	y
	a	b	c	d	d	n	m	p	t	y
	a	b	c	d	d	m	n	p	t	y

3.5 Tehnica sortării prin diminuarea incrementului – Shell

➢Aplicații care utilizează Shell Sort:



- µClibc – mică bibliotecă C standard
  - pentru sisteme de operare bazate pe Linux dedicate sistemelor embedded și dispozitivelor mobile
  - potrivit pentru microcontrolere
  - free, open-source

- bzip2 – program de compresie a fișierelor
  - free, open-source

bzip2

3.6 Tehnica sortării prin partiționare – quickSort

- ➢Creșterea eficienței metodei de sortare prin interschimbare se poate obține mărind distanța pe care se face deplasarea elementelor

- ➢Recapitulare – tehnica sortării prin interschimbare:

d	a	m	x	b	d	n
d	a	m	b	x	d	n
d	a	b	m	x	d	n
a	d	b	m	x	d	n
a	d	b	m	d	x	n
a	d	b	d	m	x	n
a	b	d	d	m	x	n
a	b	d	d	m	n	x
a	b	d	d	m	n	x
a	b	d	d	m	n	x
a	b	d	d	m	n	x

3.6 Tehnica sortării prin partiționare – quickSort

➢ Principiul metodei (1):

- se consideră tabloul de sortat $a_1, a_2, \ldots, a_n$
- se selectează un element oarecare x (pivot)
- se parcurge tabloul de la stânga la dreapta până la întâlnirea primului element $a_i > x$
- se parcurge tabloul de la dreapta la stânga până la întâlnirea primului element $a_j < x$
- se interschimbă a_i cu a_j și procesul continuă în aceeași manieră din punctul în care a rămas, până când parcurgerile se întâlnesc

```
repeta
  - cauta primul element
  a[i]>=x prin parcurgere
  stanga -> dreapta
  - cauta primul element
  a[j]<=x prin parcurgere
  dreapta -> stanga
  - daca i<=j atunci
      • interschimba a[i]
        cu a[j]
pana cand parcurgerile se
intalnesc (i>j)
```

3.6 Tehnica sortării prin partiționare – quickSort

➢ Principiul metodei (2):

- în urma acestui proces tabloul este împărțit în două partiții:
  - partiția stângă cu elemente <x
  - partiția dreaptă cu elemente >x
- se aplică aceeași procedură celor două partiții astfel rezultate
  - până la partiții banale de câte 1 element

3.6 quickSort – alegerea pivotului

- primul element
  - este o alegere acceptabilă dacă elementele se află inițial într-o ordine aleatorie
  - dacă tabloul este deja sortat sau elementele sunt în ordine inversă, va rezulta practic o slabă partiționare
- un element la întâmplare
- un element rezultat ca mediană unui grup de trei elemente
  - mediana unui grup de N numere – este cel mai mare în raport cu N/2 numere
  - alegerea celor trei elemente la extreme și mijloc – elimina cazul defavorabil al unei intrări gata sortate
- uzual, elementul situat ca poziție la mijlocul intervalului

3.6 quickSort – implementare recursivă

```
void qsR(char *s, int st, int dr){
int i=st, j=dr; char x, y;
     x = s[(st+dr)/2];
     do{
          while(s[i]<x && i<dr) i++;
          while(x<s[j] && j>st) j--;
          if(i<j){
               y=s[i];s[i]=s[j];s[j]=y;
               i++;j--;
          }
     }while (i<=j);
     if(st<j) qsR(s, st, j);
     if(i<dr) qsR(s, i, dr);
}
```

Apel:

```
void quickSort(char *s, int n){
     qsR(s, 0, n-1);
}
```

3.6 quickSort – implementare iterativă

➢ Recursivitatea este substituită printr-o iterație: toate partițiile amânate ca procesare sunt menținute într-o stivă

```
- se introduce intervalul inițial de sortat în stivă (limitele acestuia - push)
- repetă
      - se extrag din stivă limitele intervalului => interval curent (pop)
      - repetă
            - se partiționează intervalul curent
      până când terminare partiționare
      - dacă există interval stâng atunci
            - se introduc limitele sale în stivă (push)
      - dacă există interval drept atunci
            - se introduc limitele sale în stivă (push)
până când stiva se golește
```

3.6 quickSort – implementare iterativă

```
void qsI(char *s, int st, int dr){
int i, j; char x, y;
int stiva[dr-st+1]; //creare stiva
int top=-1; //initializare varf
  stiva[++top]=st; //depune stanga
  stiva[++top]=dr; //depune dreapta
  while(top >= 0){ //cat timp stiva nu e goala
    dr=stiva[top--]; //extrage dreapta
    st=stiva[top--]; //extrage stanga
    x=s[(st+dr)/2]; i=st; j=dr;
    do{
      while(s[i]<x && i<dr) i++;
      while(x<s[j] && j>st) j--;
        if(i<=j){
          y=s[i];s[i]=s[j];s[j]=y;
          i++;j--;
        }
    }while (i<=j);
    if(st<j){
      /*salvare limite
      partitie stanga*/
      stiva[++top]=st;
      stiva[++top]=j;
    }
    if(i<dr){
      /*salvare limite
      partitie dreapta*/
      stiva[++top]=i;
      stiva[++top]=dr;
    }
  }//de la while
}//de la functie
```

Apel:

```
void quickSort(char *s, int n){
      qsI(s, 0, n-1);
}
```

3.6 quickSort – analiză

- Comparații pentru o partiționare C_0
  - **C_0=n** – după alegerea unui pivot x sunt necesare n comparații
- Mișcări pentru o partiționare M_0
  - se determină prin metode probabilistice
  - presupunând că elementele partiției curente au valorile 1,2 ,…, n și pivotul ales are valoarea x, probabilitatea ca un element să fie >=x este (n-x+1)/n
  - numărul de schimbări necesare: (n-x)(n-x+1)/n
  - deoarece x poate fi ales ca orice valoare 1, 2,…, n, va rezulta un număr mediu de mișcări realizate pentru o partiționare, aproximat la: **M_0=n/6**
- Presupunând că de fiecare dată în procesul de partiționare va fi selectat drept pivot mediana (elementul situat ca valoare în mijlocul partiției, când aceasta este ordonată), atunci numărul **maxim de treceri** va fi **$\log_2 n$**
  - C = $n\log_2 n$
  - M = $(n/6) \log_2 n$ (**cazul optim**)

3.6 quickSort – analiză

- În **situații reale**, performanța medie este cu aproximativ 40% inferioară performanței optime
  - performanțe moderate pentru valori mici ale lui n

- Algoritmul se comportă straniu:
  - situația cea mai favorabilă o reprezintă tabloul inițial ordonat invers
  - situația cea mai defavorabilă, tabloul inițial ordonat

- În situația în care la fiecare partiționare se alege ca pivot elementul cu valoare **extremă** (cel mai mare sau cel mai mic)
  - performanta degenerează la $O(n^2)$
  - n partiționări în loc de $\log_2 n$

3.6 quickSort – unde e folosit?

- E potrivit atunci când poate fi folosită o metodă recursivă
- E recomandat când performanțele contează
- Fiind o tehnică divide-et-impera (divide-and-conquer) permite paralelizarea execuției

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

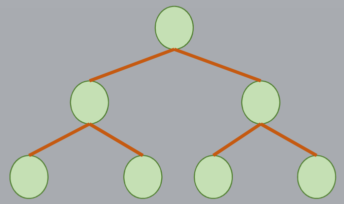
- Generalizează metoda sortării prin selecție
- Recapitulare – metoda sortării prin selecție:

d	a	m	x	b	d	n
a	d	m	x	b	d	n
a	b	m	x	d	d	n
a	b	d	x	m	d	n
a	b	d	d	m	x	n
a	b	d	d	m	x	n
a	b	d	d	m	n	x

- Demonstrație heapSort
  - https://www.youtube.com/watch?v=MtQL_Il5KhQ

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- La o primă trecere prin cele n elemente de sortat se va determina prin n/2 comparații **cel mai mic element al fiecărei perechi**
- În următoarea trecere, prin n/4 comparații se determina cel mai mic element din perechile formate din elementele selectate în pasul precedent
- Cele n/2+n/4+….+1=n-1 comparații vor conduce la un **arbore de selecție** având ca rădăcină elementul minim
- Sortarea constă în extragerea minimului din rădăcina arborelui de selecție și refacerea acestuia, ceea ce va conduce la un nou minim în rădăcină, care la rândul sau poate fi eliminat
- După n astfel de pași, arborele de selecție devine vid și procesul de sortare este încheiat
- $O((n\text{-}1) + n\log_2 n) \rightarrow \mathbf{O(n\log_2 n)}$



3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- Dezavantaje ale arborelui de selecție:
  - Arborele utilizat pentru sortarea a n elemente necesită 2n-1 locații de memorie
  - În procesul de sortare, rămân locații lipsite de informații care sunt sursa unor comparații inutile
  - Nu se poate realiza o sortare „in situ”, necesitând o structură de date suplimentară pentru păstrarea elementelor sortate

- Aceste dezavantaje sunt eliminate în structura de date numită ansamblu sau heap:
- Un ansamblu (heap) este definit drept o secvență de elemente
  - $h_s, h_{s+1}, h_{s+2}, \dots, h_d$
  - astfel încât $h_i <= h_{2i}$ și $h_i <= h_{2i+1}$ pentru orice $i=s \dots d/2$

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- Reprezentarea ansamblului cu ajutorul structurii de date tablou presupune etapele:
  - se numerotează elementele ansamblului, nivel cu nivel, de sus în jos și de la stânga la dreapta
  - se asociază elementelor ansamblului locațiile unui tablou h[n], astfel încât elementului h_i al ansamblului îi va corespunde locația h[i] a tabloului
  - în cazul sortării în ordine crescătoare, în locația h[1] se va afla elementul de valoare minimă
- Construcția ansamblului pornește de la observația:
  - dacă se dispune de ansamblul h_{s+1}, h_{s+2}, ..., h_d se poate obține ansamblul extins h_s, h_{s+1}, ..., h_d, dacă elementul adăugat se plasează în rădăcina arborelui, fiind apoi deplasat „în jos" spre locul lui, de-a lungul drumului indicat de componentele cele mai mici
  - această manieră de deplasare „în jos" conservă caracteristicile care definesc un ansamblu

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- Se consideră tabloul h în care se introduc elementele din care se dorește a se construi ansamblul
- Elementele plasate începând cu indicele n/2 formează deja un ansamblu deoarece nu exista nicio pereche de indici i și j care să satisfacă relația j=2i sau j=2i+1
  - aceste elemente formează structura de bază a ansamblului („frunzele” arborelui binar asociat)
- Ansamblul este extins spre stânga cu câte un element la fiecare pas, până când se ajunge la prima poziție a tabloului
- În fiecare pas, elementul nou introdus migrează la locul potrivit conform unei proceduri de deplasare

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

```
deplasare(s, d):
i=s; /*i precizeaza nivelul curent in ansamblu*/
j=2*i; /*j precizeaza nivelul urmator*/
x=h[i];
cat timp exista niveluri in ansamblu (j<=d) si locul nu a fost
gasit executa
     - selecteaza drept elem. curent cel mai mic elem. de pe
nivelul urmator j (h[j] sau h[j+1])
     - daca x > elem. curent atunci
          - se deplaseaza elem curent de pe nivelul urmator pe
cel curent (h[i] ← h[j])
          - avanseaza parcurgerea la nivelul urmator (i=j;
j=2*i;)
     altfel
          return (locul a fost gasit)
se plaseaza x la locul lui (h[i] ← x)
```

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- **Construcția** ansamblului „in situ” necesită repetarea procedurii deplasare pentru toate cele n/2 elemente în afara șirului de bază:

```
s=(n/2)+1;
while (s>1){
        s=s-1; deplasare(s,n);
}
```

- **Sortarea** heapsort „in situ” presupune execuția a n pași de deplasare, după fiecare pas selectându-se elementul de pe poziția 1 (vârful ansamblului) care se interschimbă cu elementul x de pe poziția n
- Se reduce dimensiunea ansamblului cu 1 la dreapta și se lasă componenta x să „migreze” la locul potrivit aplicând procedura de deplasare:

```
d=n;
while(d>1){
        h[1]<->h[d]
        d=d-1;
        deplasare(1,d);
}
```

3.7 heapSort – analiză

- **Construcția** ansamblului se realizează în n/2 pași, fiecare pas deplasând elemente de-a lungul a $\log_2(n/2)$ poziții, respectiv $\log_2((n/2)+1)$… $\log_2(n-1) \rightarrow O(n\log_2 n)$
- **Sortarea** necesită n-1 deplasări cu cel mult $\log_2(n-1)$, $\log_2(n-2)$, …,1 mișcări
- Mai sunt necesare n-1 mișcări pentru a plasa elementele sortate începând cu indicele 1, elementul minim
- $O((n/2)\log_2(n-1)+(n-1)\log_2(n-1)+(n-1))$ C $\rightarrow$ **$O(n\log_2 n)$**
- Numărul mediu de mișcări: $M=(n\log_2 n)/2$ (determinat statistic)
- Este dificil de stabilit cazul cel mai defavorabil
- Metoda heapsort **este eficientă dacă elementele de sortat sunt inițial apropiate de ordonarea inversă**

3.8 Tehnica sortării binSort

- Procesul de sortare poate fi mult accelerat dacă sunt **informații apriorice** asupra elementelor de sortat
- Se presupune că se lucrează cu un set de chei de tip întreg aparținând intervalului [0,n-1], fără duplicate (sunt distincte)
  - elementele tabloului sunt de fapt permutări ale indicilor
- Dacă se renunță la restricția „in situ" se vor utiliza **două tablouri** de dimensiunea n, a și b, sortarea elementelor din a în b realizându-se într-o singura trecere prin plasarea fiecărui element la locul potrivit în b:
  - b[i] va fi locul cheii având valoarea i
- Metoda poate fi realizata și „in situ" prin utilizarea unor **variabile** pentru păstrarea valorilor până la găsirea locului
- Procedura de sortare constă deci în verificarea fiecărei chei și introducerea ei în locul (*bin*-ul) corespunzător

3.8 Tehnica sortării binsort

- O altă metodă (tot „in situ”) urmărește ca într-o anumită etapă i
  - cât timp pe poziția i nu se găsește valoarea i, ci o valoare j≠i, se interschimbă valorile de pe pozițiile i și j
  - v[i] ↔ v[v[i]]
- Performanța: O(n)
- Constrângeri:
  - domeniu limitat
  - chei unice
- Dacă se acceptă și chei identice (m=nr de chei), performanța scade: O(n+m)

tablou: 5 3 0 4 6 2 1

Etapa								
0	ind	0	1	2	3	4	5	6
	val	5	3	0	4	6	2	1
	ind	0	1	2	3	4	5	6
	val	2	3	0	4	6	5	1
1	ind	0	1	2	3	4	5	6
	val	0	3	2	4	6	5	1
	ind	0	1	2	3	4	5	6
	val	0	4	2	3	6	5	1
	ind	0	1	2	3	4	5	6
	val	0	6	2	3	4	5	1
2, 3, ...	ind	0	1	2	3	4	5	6
	val	0	1	2	3	4	5	6

3.9 Tehnica sortării prin determinarea distribuțiilor

- Se consideră un tablou de n elemente, cu cheile cuprinse în intervalul [0, m-1]
- Dacă m<n rezultă că tabloul va conține și elemente identice
- Ideea algoritmului de sortare este:
  - de a contoriza într-o primă trecere numărul de elemente pentru fiecare valoare posibilă de cheie (distribuțiile)
  - iar apoi într-o a doua trecere de a muta elementele direct în poziția lor ordonată
- Sunt necesare **două structuri de date auxiliare**: un tablou pentru contorizarea distribuțiilor și un tablou în care se va păstra structura sortată
- Performanța operației de sortare: O(n)

Universitatea
Politehnica
Timişoara

P-ţa Victoriei nr. 2
RO 300006 - Timişoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Vă mulțumesc!