1. Základné pojmy:

Údaje.
Údaje predstavujú abstrakciu reálneho sveta, vyjadrujú sa nimi určité charakteristiky a vlastnosti reálnych objektov.
Statické údaje
Počet prvkov údaja sa po vykonaní operácie nad ním nemení.
Dynamické údaje
Počet prvkov údaja sa po vykonaní operácie nad ním môže zmeniť.
Algoritmus.
Algoritmus určuje spôsob transformácie vstupných údajov na výstupe, pričom transformácia sa uskutočňuje konštruktívnym spôsobom v diskrétnych krokoch. V každom kroku sa uskutočňuje akcia trvajúca konečný čas, ktorá prispieva do celkovej transformácie, celkového riešenia. Algoritmus predpisuje takú postupnosť akcií, ktorá predpisuje riešenie daného problému.
Typ údajov
1. Typ údajov určuje množinu hodnôt, do ktorej daná konštanta prináleží, alebo ktoré môže daná premenná či výraz nadobudnúť, alebo ktoré môže daným operátorom alebo funkciou vypočítať.
2. Typ hodnoty konštanty, premennej alebo výrazu sa dá určiť z ich deklarácie alebo zápisu bez nevyhnutnosti uskutočniť vlastný výpočtový proces programu.
3. Každý operátor alebo funkcia predpokladá argumenty presne definovaných typov a poskytuje výsledok tiež presne definovaného typu. V prípade, že operátor pripúšťa argumenty rôznych typov, dá sa typ výsledku určiť podľa špecifických pravidiel príslušného programovacieho jazyka.
Štruktúra údajov
An organization of information, usually in memory, for better *algorithm* *efficiency*, such as *queue*, *stack*, *linked list*, *heap*, *dictionary*, and *tree*, or conceptual unity, such as the name and address of a person. It may include redundant information, such as length of the *list* or number of *nodes* in a *subtree*.
údajové typy sa delia na

- *jednoduché (primitívne) údajové typy*
  - Prirodzené čísla, Celé čísla, Reálne čísla, Logické hodnoty, Znaky
- *zložené (štruktúrované) typy údajov*
  - Pole, Reťazec, Vektor, Množina, Zásobník, Graf, Tabuľka, Strom, Zoznam, Záznam, Front, Zreťazená voľná pamäť

Implementujúce typy údajov.
Implementujúcim typom údajov je vektor bitov. Záleží len na tom, akú interpretáciu mu prisúdime.
Môžeme ho považovať napr. za:
* zápis čísla v pevnej rádovej čiarke
* zápis čísla v pohyblivej rádovej čiarke
* zápis boolovskej hodnoty
* obraz znaku vonkajšej abecedy v danom kóde

Dôležitým parametrom je *rozmer* vektora bitov. Pri voľbe rozmeru musíme zohľadniť jednak požiadavky, ktoré vyplývajú z možného rozsahu údajov, a jednak požiadavky efektívnosti implementácie základných operácií.
Ak je rozsah údajov daného typu taký, že môže byť až 2 na k rôznych údajov, musíme zvoliť rozmer vektora bitov aspoň k. Rozmer vektora však musíme prispôsobiť aj vlastnostiam daného počítača. Ak je napr. jeho pamäť štruktúrovaná tak, že jedno slovo pozostáva z n bitov, je užitočné voliť rozmer vektora bitov ako malý celočíselný násobok čísla n. Ak sú na danom počítači realizované základné operácie nad daným typom údajov ako strojové, je užitočné použiť ich a podľa požiadavky voliť aj rozmer. Ak sú napr. na danom počítači realizované základné aritmetické operácie nad danými celými číslami ako strojové, pričom ich operandy sú vždy jednoslovné, z hľadiska efektívnej implementácie je vhodné voliť reprezentáciu celých čísel v jednom slove. Tým je potom spätne určený *rozsah použiteľných* čísel. Napr. pri minipočítačoch je rozsah pri použitom doplnkovom kóde <-32 768, 32767>.

Dátový typ
abstract data type
A set of data values and associated operations that are precisely specified independent of any particular implementation.

Špecifikácia
Operačná špecifikácia – je priama realizácia štruktúr údajov a prípustných hodnôt operácii prekladačom použitého programovacieho jazyka.
Axiomatická špecifikácia
Defining the behavior of an abstract data type with axioms.

EXAMPLE:
Operácie

push:	stack, elm	-> stack	pridanie prvku do zásobníka
top:	stack	-> elm	výber prvku zo zásobníka
pop:	stack	-> stack	zrušenie prvku z vrchu v zásobníku
empty:	-> stack		konštrukcia zásobníka (prázdneho)
isempty:	stack	-> bool	test, či je zásobník prázdny

Axiómy
top(push(s,e))=e
pop(push(s,e))=s
isempty(empty)=true
isempty(push(s,e))=false
pop(empty)=empty
top(empty)=error

reprezentácia
reprezentácia ľubovolného statického typu údajov je vektor
reprezentácia ľubovolného dynamického typu údajov je zreťazená voľná pamäť
reprezentácia stromov binárnym stromami
implementácia
implementácia množiny logickým poľom
implementácia množiny binárnym vyhľadávacim stromom (BVS)
implementácia množiny vyváženým stromom
implementácia zásobníka pomocou poľa
implementácia zásobníka pomocou zreťazenej voľnej pamäti

zapuzdrenie information hiding and separation of concerns, in software engineering, the process of enclosing programming elements inside larger, more abstract entities. Information hiding reduces software development risk by shifting the code's dependency on an uncertain implementation (design decision) onto a well-defined interface. Clients of the interface perform operations purely through the interface so if the implementation changes, the clients do not have to change.

Implementovaný typ a implementujúci typ viď vyššie.

2. Spájaný zoznam

http://stsdas.stsci.edu/bps/linked_list.html

Sémantiku operácií môžeme popísať takto:

empty - vytvorenie prázdneho zoznamu,
first - (začiatok zoznamu) zoznam, ktorý bol sprístupnený prostredníctvom niektorého jeho (ľubovoľného) prvku, bude prístupný prostredníctvom prvého prvku,
last - (koniec zoznamu) zoznam, ktorý bol sprístupnený prostredníctvom niektorého jeho (ľubovoľného) prvku, bude prístupný prostredníctvom jeho posledného prvku,
next - (nasledujúci prvok) zoznam, ktorý bol sprístupnený prostredníctvom niektorého jeho (ľubovoľného) prvku, bude prístupný prostredníctvom nasledujúceho prvku,
previous - (predchádzajúci prvok) zoznam, ktorý bol sprístupnený prostredníctvom niektorého jeho (ľubovoľného) prvku, bude prístupný prostredníctvom predchádzajúceho prvku,
read - vrátenie hodnoty toho prvku zoznamu, prostredníctvom ktorého je zoznam sprístupnený,
P-insert - vloženie nového prvku pred prvok, prostredníctvom ktorého je zoznam sprístupnený,
N-insert - vloženie nového prvku za prvok, prostredníctvom ktorého je zoznam sprístupnený,
P-delete - zrušenie toho prvku zoznamu, prostredníctvom ktorého je zoznam sprístupnený a sprístupnenie zoznamu prostredníctvom predchádzajúceho prvku,
N-delete -zrušenie toho prvku zoznamu, prostredníctvom ktorého je zoznam sprístupnený a sprístupnenie zoznamu prostredníctvom nasledujúceho prvku,
isempty - test, či je zoznam prázdny.

Reprezentácia zoznamu musí zahrnovať dve veci:

- reprezentáciu samotného zoznamu, t.j. všetkých prvkov v danej postupnosti,
– reprezentáciu pozície prvku, prostredníctvom ktorého je zoznam prístupný.

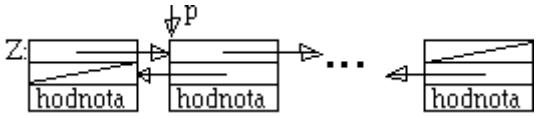


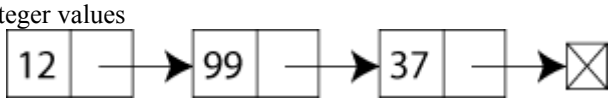
Schéma reprezentácie zoznamu,
Z - samotný zoznam, P - pozícia sprístupnenia.

Konkrétny spôsob implementácie závisí, od voľby implementujúceho typu údajov. Veľmi elegantná je implementácia v jazyku pascal s použitím smerníkov. Implementujúci typ je v tomto prípade zreťazená voľná pamäť jazyka pascal.

Singly-linked list

The simplest kind of linked list is a singly-linked list (or slist for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

A singly linked list containing three integer values



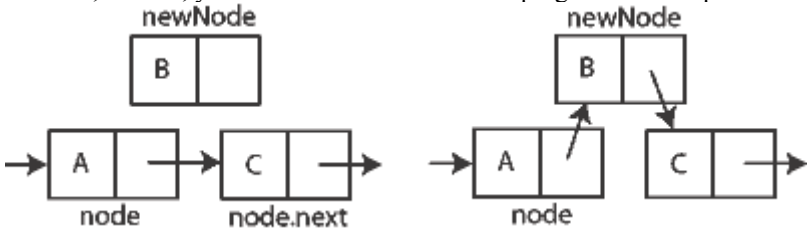
Singly-linked lists
Our node data structure will have two fields. We also keep a variable *firstNode* which always points to the first node in the list, or is *null* for an empty list.

```
record Node {
    data // The data being stored in the node
    next // A reference to the next node; null for last node
}

record List {
    Node firstNode   // points to first node of list; null for empty list
}
```

Traversal of a singly-linked list is easy, beginning at the first node and following each *next* link until we come to the end:

```
node := list.firstNode
while node not null {
    (do something with node.data)
    node := node.next
}
```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done; instead, you have to locate it while keeping track of the previous node.




```
function insertAfter(Node node, Node newNode) { // insert newNode after node
    newNode.next := node.next
    node.next    := newNode
}
```

Inserting at the beginning of the list requires a separate function. This requires updating *firstNode*.

```
function insertBeginning(List list, Node newNode) { // insert node before current first
node
    newNode.next   := list.firstNode
    list.firstNode := newNode
}
```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.

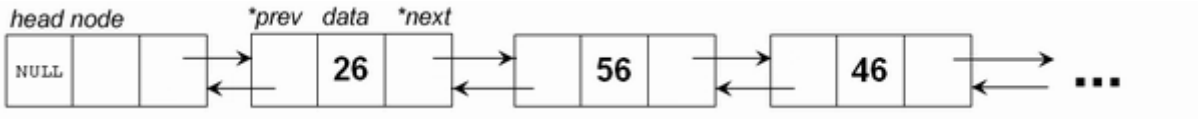
```
function removeAfter(Node node) { // remove node past this one
    obsoleteNode := node.next
    node.next := node.next.next
    destroy obsoleteNode
```

```
}

function removeBeginning(List list, Node node) { // remove first node
    obsoleteNode := list.firstNode
    list.firstNode := list.firstNode.next          // point past deleted node
    destroy obsoleteNode
}
```

Notice that removeBeginning() sets *list.firstNode* to *null* when removing the last node in the list.
Since we can't iterate backwards, efficient "insertBefore" or "removeBefore" operations are not possible.

Doubly-linked list

A more sophisticated kind of linked list is a doubly-linked list or two-way linked list. Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node.



An example of a doubly linked list.

In some very low level languages, Xor-linking offers a way to implement doubly-linked lists using a single word for both links, although the use of this technique is usually discouraged.

With doubly-linked lists there are even more pointers to update, but also less information is needed, since we can use backwards pointers to observe preceding elements in the list. This enables new operations, and eliminates special-case functions. We will add a *prev* field to our nodes, pointing to the previous element, and a *lastNode* field to our list structure which always points to the last node in the list. Both *list.firstNode* and *list.lastNode* are *null* for an empty list.

```
record Node {
   data // The data being stored in the node
   next // A reference to the next node; null for last node
   prev // A reference to the next node; null for first node
}

record List {
    Node firstNode   // points to first node of list; null for empty list
    Node lastNode    // points to final node of list; null for empty list
}
```

Iterating through a doubly linked list can be done in either direction. In fact, direction can change many times, if desired.

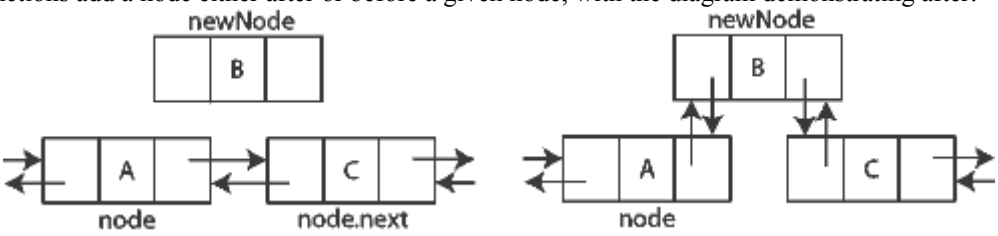
Forwards

```
node := list.firstNode
while node ≠ null
    <do something with node.data>
    node := node.next
```

Backwards

```
node := list.lastNode
while node ≠ null
    <do something with node.data>
    node := node.prev
```

These symmetric functions add a node either after or before a given node, with the diagram demonstrating after:




```
function insertAfter(List list, Node node, Node newNode)
    newNode.prev := node
    newNode.next := node.next
    if node.next = null
        list.lastNode := newNode
    else
        node.next.prev := newNode
    node.next := newNode
function insertBefore(List list, Node node, Node newNode)
    newNode.prev := node.prev
```

```
    newNode.next := node
    if node.prev is null
        list.firstNode := newNode
    else
        node.prev.next := newNode
    node.prev     := newNode
```

We also need a function to insert a node at the beginning of a possibly-empty list:

```
function insertBeginning(List list, Node newNode)
    if list.firstNode = null
        list.firstNode := newNode
        list.lastNode  := newNode
        newNode.prev := null
        newNode.next := null
    else
        insertBefore(list, list.firstNode, newNode)
```

A symmetric function inserts at the end:

```
function insertEnd(List list, Node newNode)
    if list.lastNode = null
        insertBeginning(list, newNode)
    else
        insertAfter(list, list.lastNode, newNode)
```

Removing a node is easier, only requiring care with the *firstNode* and *lastNode*:

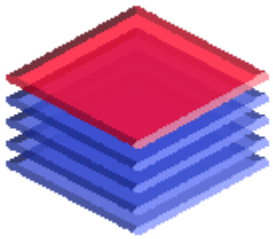
```
function remove(List list, Node node)
  if node.prev = null
      list.firstNode := node.next
  else
      node.prev.next := node.next

  if node.next = null
      list.lastNode := node.prev
  else
      node.next.prev := node.prev
  destroy node
```

One subtle consequence of this procedure is that deleting the last element of a list sets both *firstNode* and *lastNode* to *null*, and so it handles removing the last node from a one-element list correctly. Notice that we also don't need separate "removeBefore" or "removeAfter" methods, because in a doubly-linked list we can just use "remove(node.prev)" or "remove(node.next)" where these are valid.

Zásobník

Druhy:

* stack, elm, bool



Operácie:

* push : stack, elm ==> stack
* top : stack ==> elm
* pop : stack ==> stack
* empty : ==> stack
* isempty : stack ==> bool

Zásobník:

Semantiku operácií môžeme opísať takto:

* empty - vytvorenie prázdneho zásobníka,
* push - pridanie prvku do zásobníka,
* top - prečítanie prvku z vrchu zásobníka,
* pop - zrušenie prvku z vrchu zásobníka,
* isempty - test, ci je zásobnik prázdny.

Zásobník možno implementovať pomocou :
poľa / zreťazenej voľnej pamäti
Vzhľadom na to, že ide o dynamický typ údajov, je prirodzenejšie voliť ako implementujúci typ **zreťazenú voľnú pamäť.**

Definition: A collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop. Often top and isEmpty are available, too. Also known as "last-in, first-out" or LIFO.

Formal Definition: The operations new(), push(v, S), top(S), and popoff(S) may be defined with *axiomatic semantics* as follows.

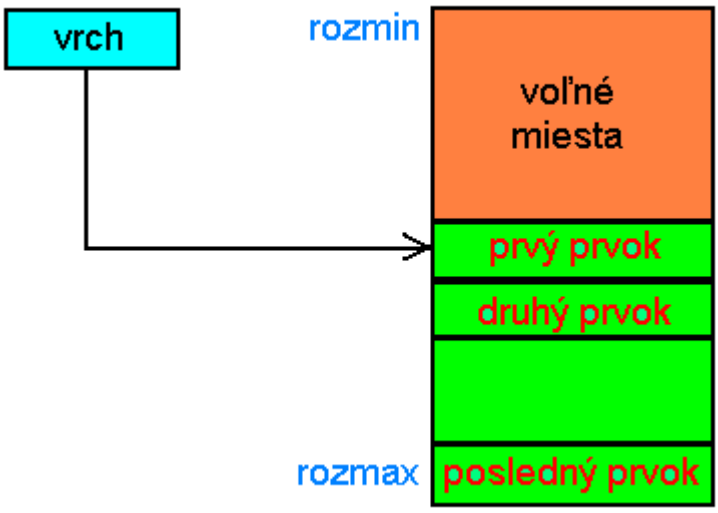
1. new() returns a stack
2. popoff(push(v, S)) = S
3. top(push(v, S)) = v

where S is a stack and v is a value. The pop operation is a combination of top, to return the top value, and popoff, to remove the top value.

The *predicate* isEmpty(S) may be defined with the following additional axioms.

4. isEmpty(new()) = true
5. isEmpty(push(v, S)) = false

Also known as LIFO.



3.b Front

Typ údajov front je príbuzný typu údajov zásobník . Spôsob práce s frontom možno charakterizovať vetou "prvý dnu, prvý von". Označuje sa názvom FIFO (First In First Out).

* špecifikácia typu údajov front,
* implementácia pomocou poľa,
* implementácia pomocou zreťazenej voľnej pamäti.

V praxi sa stretávame aj typom údajov, ktorý je istou modifikáciou frontu v tom, že pridávanie a výber prvkov možno robiť na obidvoch koncoch. Niekedy sa takémuto typu údajov hovorí balík alebo obojsmerný front.

Špecifikácia typu front.

Druhy:

* front
* elm
* bool

Operácie:

* insert : front, elm ==> front
* remove : front ==> front
* first : front ==> elm
* empty : ==> front
* isempty : front ==> bool

Front je špecifikovaný signatúrou na obrázku:

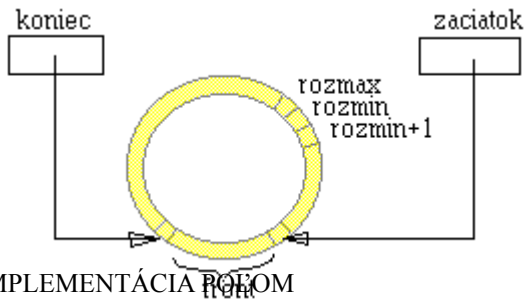
Front.

Signatúra typu údajov front.

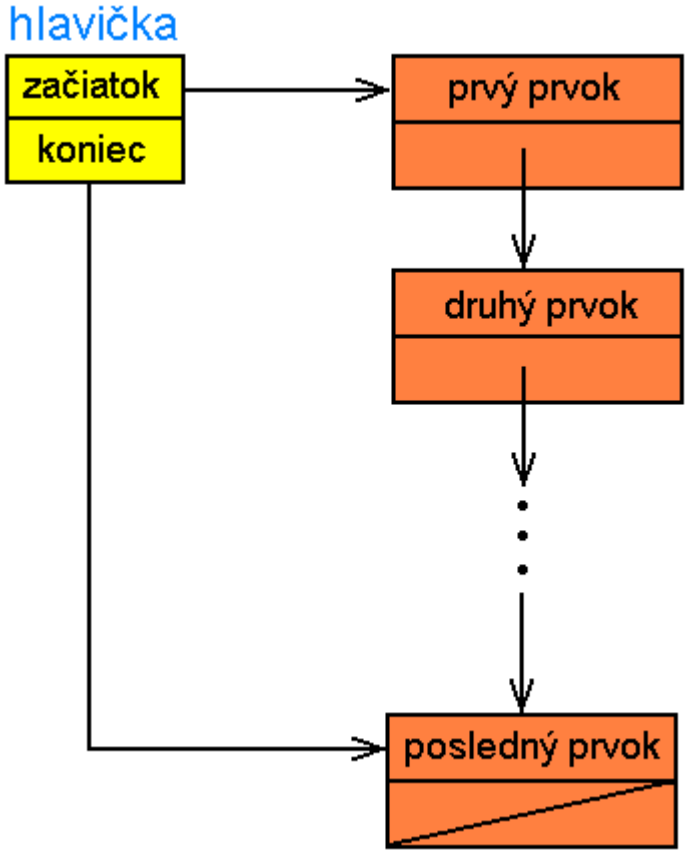
Sémantiku operácií môžeme opísať takto:

* empty - vytvorenie prázdneho frontu,
* insert - pridanie prvku do frontu,
* first - prečítanie prvku z frontu,
* remove - zrušenie prvku vo fronte,
* isempty - test, či je front prázdny.

Front možno implementovať pomocou



IMPLEMENTÁCIA POĽOM



poľa a zreťazenej voľnej pamäti
implementácia zreťazenou voľnou pamätou

front / queue
(data structure)

Definition: A collection of items in which only the earliest added item may be accessed. Basic operations are add (to the *tail*) or enqueue and delete (from the *head*) or dequeue. Delete returns the item removed. Also known as "first-in, first-out" or FIFO.

Formal Definition: It is convenient to define delete or dequeue in terms of remove and a new operation, front. The operations new(), add(v, Q), front(Q), and remove(Q) may be defined with *axiomatic semantics* as follows.

1. new() returns a queue
2. front(add(v, new())) = v
3. remove(add(v, new())) = new()
4. front(add(v, add(w, Q))) = front(add(w, Q))
5. remove(add(v, add(w, Q))) = add(v, remove(add(w, Q)))

where Q is a queue and v and w are values.

Also known as FIFO

PREVODY MEDZI INFIXOM, PREFIXOM a POSTFIXOM
Prevody aritmetických výrazov medzi Infixom, Postfixom a Prefixom
Infix -> Postfix
str – načítaný postfix, posúvanie sa v rámci str pomocou ukazovateľa s (na začiatku nastavený na začiatok str)
target – výstupný výraz v infixe, posúvanie sa v rámci target pomocou ukazovateľa t (na začiatku nastavený na začiatok target)
stack – zásobník znakov

priorita ‘+‘, ‘-, je 1
priorita ‘*‘, ‘/, je 2

pokiaľ s nie je na konci str rob
1. ak *s je operand, tak do *t = *s, zvýš oba ukazovatele
2. ak *s je ‘(‘, tak push(stack, *s), zvýš ukazovateľ s
3. ak *s je ‘)‘, tak n = top(stack), pop(stack)

pokiaľ n != ‘(‘ rob *t = n, zvýš ukazovateľ t, n = top(stack), pop(stack)

4. ak s je **‘+’, ‘-‘,** ‘*’, ‘/’, tak

ak je zásobník prázdny, tak push(stack, *s),

inak n = top(stack), pop(stack),
pokiaľ je priorita n >= priorita *s rob
*t = n, zvýš ukazovateľ t, n = top(stack), pop(stack)
push(stack, n)
push(stack, *s)
zvýš ukazovateľ s

Infix -> Prefix

str – načítaný postfix, posúvanie sa v rámci str pomocou ukazovateľa s (na začiatku nastavený na začiatok str)
target – výstupný výraz v infixe, posúvanie sa v rámci target pomocou ukazovateľa t (na začiatku nastavený na koniec target)
stack – zásobník znakov

priorita ‘+‘, ‘-, je 1
priorita ‘*‘, ‘/, je 2

pokiaľ s nie je na konci str rob
1. ak *s je operand, tak do *t = *s, zvýš ukazovateľ s a zníž ukazovateľ t
2. ak *s je ‘)‘, tak push(stack, *s), zvýš ukazovateľ s
3. ak *s je ‘(‘, tak n = top(stack), pop(stack)

pokiaľ n != ‘)‘ rob *t = n, zníž ukazovateľ t, n = top(stack), pop(stack)

4. ak s je **‘+’, ‘-‘, ‘***’, ‘/’, tak

ak je zásobník prázdny, tak push(stack, *s),
inak n = top(stack), pop(stack),
pokiaľ je priorita n >= priorita *s rob
*t = n, zníž ukazovateľ t, n = top(stack), pop(stack),
push(stack, n)
push(stack, *s)
zvýš ukazovateľ s

Postfix -> Prefix

str – načítaný postfix, posúvanie sa v rámci str pomocou ukazovateľa s (na začiatku nastavený na začiatok str)
stack – zásobník stringov (reťazcov znakov)

pokiaľ s nie je na konci str rob

1. ak *s je **‘+’, ‘-‘, ‘***’, ‘/’, tak

n1 = top(stack), pop(stack)
n2 = top(stack), pop(stack)
do x spoj *s, n2, n1
push(stack, x)
inak push(stack, *s)

2. zvýž ukazovateľ s

vypíš obsah stacku

Prefix -> Postfix

str – načítaný postfix, posúvanie sa v rámci str pomocou ukazovateľa s (na začiatku nastavený na koniec str)
stack – zásobník stringov (reťazcov znakov)

pokiaľ s nie je na začiatku str rob

1. ak s je **‘+’, ‘-‘, ‘***’, ‘/’, tak

n1 = top(stack), pop(stack)
n2 = top(stack), pop(stack)
do x spoj n2, n1, *s
push(stack, x)
inak push(stack, *s)

2. zníž ukazovateľ s

vypíš obsah stacku

Prefix -> Infix

str – načítaný postfix, posúvanie sa v rámci str pomocou ukazovateľa s (na začiatku nastavený na koniec str)
stack – zásobník stringov, reťazcov znakov

pokiaľ s nie je na začiatku str rob
1. ak s je **‘+’, ‘-‘, ‘*’**, ‘/’, tak
n1 = top(stack), pop(stack)
n2 = top(stack), pop(stack)
do x spoj ‘(‘,n2, n1, *s, ‘)‘
push(stack, x)
inak push(stack, *s)
2. zníž ukazovateľ s
vypíš obsah stacku

Postfix -> Infix

str – načítaný postfix, posúvanie sa v rámci str pomocou ukazovateľa s (na začiatku nastavený na začiatok str)
stack – zásobník stringov, reťazcov znakov

pokiaľ s nie je na konci str rob
1. ak s je **‘+’, ‘-‘, ‘*’**, ‘/’, tak
n1 = top(stack), pop(stack)
n2 = top(stack), pop(stack)
do x spoj ‘(‘,n2, n1, *s, ‘)‘
push(stack, x)
inak push(stack, *s)
2. zvýš ukazovateľ s
vypíš obsah stacku

POLE
Pole je homogénna štruktúra; pozostáva z prvkov, ktoré sú všetky jediného typu. Tento typ prvku poľa sa nazýva bázový alebo základný typ. Pole pokladáme za štruktúru s náhodným prístupom; všetky prvky môžu byť vybraté náhodne a sú rovnako sprístupnitelné. Na referenciu individuálneho prvku poľa je potrebné použiť meno celej štruktúry rozšírené o index, určujúci vybratý prvok. Index nadobúda hodnoty typu, definovaného ako typ indexu poľa.

Špecifikácia:
Druhy: array, int, elm, bool
Operácie:

* empty : int, int ==> array
* read : array, int ==> elm
* write : array, int, elm ==> array
* low : array ==> int
* high : array ==> int
* isempty : array ==> bool
* iseq : int, int ==> bool

Pole.
Signatúra typu údajov "pole".
Semantiku operácií môžeme opísať takto:

* empty - konštrukcia poľa prázdneho
* read - čítanie daného prvku poľa
* write - zápis daného prvku poľa na dané miesto
* low - dolná hranica indexu poľa
* high - horná hranica indexu poľa
* isempty - test, či je pole prázdne
* iseq - test, či sa dva indexy rovnajú

REŤAZEC
Pod reťazcom rozumieme usporiadanú n-ticu prvkov danej množiny. Počet prvkov v reťazci určuje jeho dĺžku. Reťazec s nulovou dĺžkou nazývame prázdny reťazec.
Špecifikácia:
Druhy: string, elm, bool, nat
Operácie:
* concat : string, string ==> string
* empty : ==> string
* length : string ==> nat
* insert : string, elm ==> string

```
* delmax: string          ==> string
* isin      : string, elm    ==> bool
* iseq     : string, string  ==> bool
* isempty : string        ==>  bool
```

Signatúra typu údajov reťazec.
Signatúra typu údajov "reťazec".
Popis operácií:

* concat - zreťazenie reťazcov
* empty - vytvorenie prázdneho reťazca
* length - určenie dĺžky reťazca
* insert - pridanie prvku k reťazcu
* delmax - vylúčenie maximálneho prvku
* isin - test, či je prvok v reťazci
* iseq - test, či sa dva reťazce rovnajú
* isempty - test, či je reťazec prázdny

5. Stromy

Trees are natural structures for representing certain kinds of hierarchical data.

When we say a tree, we are talking about a way to structure data. The term tree on its own does not necessarily indicate one specific way of implementing this structure. Instead it just says that however we implement it, our implementation must allow for the data to be thought of in the general way that a tree structures data

Graf je množina vrcholov pospájaných hranami. V orientovanom grafe sú hrany usporiadané dvojice vrcholov, určujú aj smer prepojenia vrcholov.

Strom je taký orientovaný graf, v ktorom

- je jeden **vrchol** špeciálny - **koreň** - do tohto vrcholu nevedie žiadna **hrana** (šípka)
- do všetkých ostatných vrcholov vedie práve jedna hrana
- graf je súvislý

Terminológia

- ak z nejakého vrcholu A vedie hrana (šípka) do vrcholu B, tak vrchol A nazveme **otcom** (alebo predchodcom) a vrchol B **synom**
- ak sú dva vrcholy spojené hranou, hovoríme, že spolu **susedia**
- ak nejaký vrchol nemá žiadnych synov, hovoríme mu **list**, inak je to **vnútorný vrchol** stromu
- **podstrom** - nejaký vrchol považujeme za koreň a všímame si len jeho synov a ich synov atď.
- **cesta** - postupnosť vrcholov spojených hranami (len jedným smerom)
- **dĺžka cesty** = počet vrcholov na ceste-1 (t.j. počet hrán na ceste)
- **výška/hĺbka** stromu = dĺžka najdlhšej cesty od koreňa k listom (prázdny strom=?; len koreň=0; ...)

Reprezentácia stromov

- ako v **halde** - v jednorozmernom poli:
  - každý prvok reprezentuje jeden vrchol stromu:
  - prvý prvok je koreň
  - í-ty vrchol má synov na indexoch 2*i a 2*i+1
  - treba evidovať "diery" - miesta, ktoré nereprezentujú vrcholy
- pole vrcholov, kde vrchol obsahuje *info* a indexy do poľa pre ľavého a pravého syna
- pole vrcholov - **každý vrchol má spájaný zoznam synov** (táto reprezentácia by sa dala použiť aj na všeobecný strom)

Všeobecné stromy

A *trie* (from re**trie**val), is a multi-way tree structure useful for storing strings over an alphabet. It has been used to store large dictionaries of English (say) words in spelling-checking programs and in natural-language "understanding" programs.

- každý vrchol má ľubovoľný počet synov

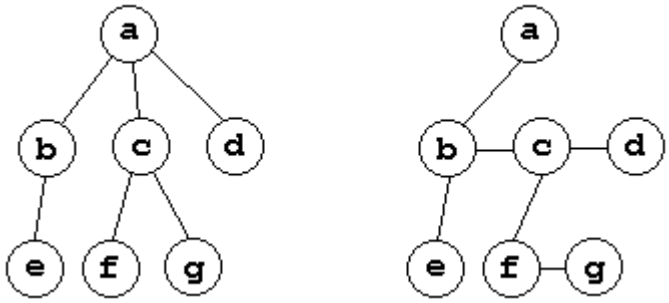
ak je N-árny, t.j. z každého vrcholu max. N vrcholov, môžeme ho reprezentovať takto:

```
type
  TStrom = class
    info:TPrvok;
    syn:array[1..N] of TStrom;
  end;
```

alebo neohraničený (dynamický) počet synov:

```
type
  TStrom = class
    info:TPrvok;
    syn:array of TStrom;
  end;
```

- Často sa v praxi používa aj iná reprezentácia - vychádza z binárneho stromu (budeme ju volať syn-brat):
  - v každom vrchole - informácia
  - 1. smerník - prvý (najľavejší, najstarší, prvorodený) syn
  - 2. smerník - brat, t.j. nasledujúci vrchol so spoločným otcom
    - reťaz vrcholov spojených smerníkom **brat** tvorí jednosmerný spájaný zoznam synov nejakého vrcholu



strom vľavo je reprezentovaný pomocou syn-brat vpravo

reprezentácia všeobecného stromu pomocou syn-brat:

```
type
  TStrom = class
    info:TPrvok;
    syn,brat:TStrom;
  end;
```

Pozn.

- koreň väčšinou nemá bratov - ak by mal, voláme túto štruktúru "**les**"

Implementation of Trees in C: //pomocou struct a pointerov

```
typedef struct _tree {
        int             data;
        struct _tree    *left, *right;
} tree_t;
```

Implementation of Trees in C: //pomocou poli

```
typedef data_t[MAX_NODES] tree_t;

/* We want to store the data from the left and the right children of node n

 * into the appropriate variables.
 */
data_t left_child, right_child;

left_child = tree[2 * n + 1];
right_child = tree[2 * n + 2];

/* Realize that we have only copied the data value, but if we modify left
 * child * or right_child, we do not change the values in the tree.  To do
 * that, we would * need to make left_child and right_child pointers to those
 * locations in the tree
 */

An inherent limitation to the array method is that cells will exist for nodes even when
there is no data at those locations. For this reason you need to put some value in empty
locations to indicate that they hold no data.
```

For example, if the data elements were positive integers, then a -1 might indicate empty. This could be done with a sharp define.

```
#define         EMPTY   -1
```

Binárny strom

Reprezentácia stromov

Types of binary tree

A **binary tree** is a **rooted** tree in which every node has at most two children.

A **full binary tree** is a tree in which every node has zero or two children.

A **perfect binary tree** is a complete binary tree in which **leaves** (vertices with zero children) are at the same **depth** (distance from the **root**, also called **height**).

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers recursively point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a **binary tree** is either empty (represented by a null pointer), or is made of a single node,

where the left and right pointers (recursive definition ahead) each point to a **binary tree**.

Typical Binary Tree Code in C/C++

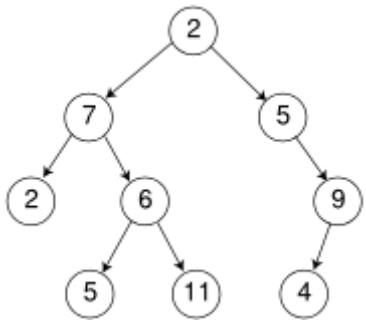
```
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```

Binárny vyhľadávací strom

Binarne vyhladavacie stromy su datove struktury podporujuce viacere operacie na dynamickych mnozinach vratane SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE. Mozno ich preto pouzit aj ako slovnik alebo ako prioritnu frontu.

Zakladne operacie na binárnom prehladavacom strome trvaju cas umerny vyske stromu. V pripade uplneho binarneho stromu s n uzlami tieto operacie trvaju v najhorsom pripade cas 0(log n).

Ak kazdy prvok v strome ma nanajvys jedneho potomka(strom je linearny),tie iste operacie trvaju v najhorsom pripade cas 0(n).



A "binary search tree" (BST) or "ordered binary tree" is a type of binary tree where the nodes are arranged in order: for each node, all elements in its left subtree are less-or-equal to the node (<=), and all the elements in its right subtree are greater than the node (>). The tree shown above is a binary search tree -- the "root" node is a 5, and its left subtree nodes (1, 3, 4) are <= 5, and its right subtree nodes (6, 9) are > 5. Recursively, each of the subtrees must also obey the binary search tree constraint: in the (1, 3, 4) subtree, the 3 is the root, the 1 <= 3 and 4 > 3. Watch out for the exact wording in the problems -- a "binary search tree" is different from a "binary tree".

Vlastnosti binarnych prehladavacich stromov:

-usporiadany binarny strom

-mozno ho reprezentovat pomocou spajanej datovej struktury,ktora v kazdom svojoom uzle obsahuje data.

-kazdy uzol obsahuje polozky:key, left, right, p.Ak potomok alebo rodic chyba, prislusna hodnota obsahuje hodnotu NULL (alebo NIL).Koren je jedinym uzlom ktoreho polozka ukazujuca na rodica obsahuje NULL.

MINVALUE

```
/*
 Given a non-empty binary search tree,
 return the minimum data value found in that tree.
```

```
 Note that the entire tree does not need to be searched.
*/
int minValue(struct node* node) {
  struct node* current = node;

  // loop down to find the leftmost leaf
  while (current->left != NULL) {
    current = current->left;
  }

  return(current->data);
}
```

MAXVALUE

```
/*
 Given a non-empty binary search tree,
 return the maximum data value found in that tree.
 Note that the entire tree does not need to be searched.
*/
int maxValue(struct node* node) {
  struct node* current = node;

  // loop down to find the rightmost leaf
  while (current->right != NULL) {
    current = current->right;
  }

  return(current->data);
}
```

printTree() Solution (C/C++)

```
/*
 Given a binary search tree, print out
 its data elements in increasing
 sorted order.
*/
void printTree(struct node* node) {
  if (node == NULL) return;

  printTree(node->left);
  printf("%d ", node->data);
  printTree(node->right);
}
```

LOOKUP

```
/*
 Given a binary tree, return true if a node
 with the target data is found in the tree. Recurs
 down the tree, chooses the left or right
 branch by comparing the target to each node.
*/
static int lookup(struct node* node, int target) {
```

```
  // 1. Base case == empty tree
  // in that case, the target is not found so return false
  if (node == NULL) {
    return(false);
  }
  else {
    // 2. see if found here
    if (target == node->data) return(true);
    else {
      // 3. otherwise recur down the correct subtree
      if (target < node->data) return(lookup(node->left, target));
      else return(lookup(node->right, target));
    }
  }
}
```

LOOKUP2

We will provide two solutions, one iterative and one recursive. The return value from both is the index in the original array. If the element is not present in the array, the sharp-defined value ~NOT_FOUND~ is returned.

```
int bin_search (int arr[], int n, int val)
{
        /* n indicates the number of cells in the array */

        int low, high, mid;

        low = 0;

        /* Set high to be the highest array index. */
        high = n - 1;

        while (high >= low) {

                /* Begin searching in the middle */
                mid = (low + high) / 2;

                /* Check if you have found it or adjust the range accordingly */
                if (arr[mid] == val) {
                        return mid;
                } else if (arr[mid] > val) {
                        high = mid - 1;
                } else {
                        low = mid + 1;
                }
        }

        return NOT_FOUND;
}
```

Now for the recursive one. The basic idea it that it keeps applying the same algorithm on the reduced range. The tricky part is offsetting the return value.

```
int bin_search (int arr[], int n, int val)
{
        int mid;

        if (n == 0) return NOT_FOUND;

        if (n == 1) return (arr[0] == val? 0 : NOT_FOUND);

        mid = (n - 1) / 2;

        /* Check if you have found it or adjust the range accordingly */
        if (arr[mid] == val) {
```

```
                return mid;
        } else if (arr[mid] > val) {
                return mid + bin_search (&arr[mid + 1], n / 2, val);
        } else {
                return mid + bin_search (&arr[mid - 1], (n - 1) / 2, val);
        }
}
```

INSERT

```
/*
 Helper function that allocates a new node
 with the given data and NULL left and right
 pointers.
*/
struct node* NewNode(int data) {
  struct node* node = new(struct node);    // "new" is like "malloc"
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}


/*
 Give a binary search tree and a number, inserts a new node
 with the given number in the correct place in the tree.
 Returns the new root pointer which the caller should
 then use (the standard trick to avoid using reference
 parameters).
*/
struct node* insert(struct node* node, int data) {
  // 1. If the tree is empty, return a new, single node
  if (node == NULL) {
    return(newNode(data));
  }
  else {
    // 2. Otherwise, recur down the tree
    if (data <= node->data) node->left = insert(node->left, data);
    else node->right = insert(node->right, data);

    return(node); // return the (unchanged) node pointer
  }
}
```

--mazanie je asi ako hladanie akurat namiesto vypisanie sa dany prvok vynuluje a uvolni sa pamat.

- no warranty.
- -compiled by PVi

10. Usporadúvanie. metódy vnútorného

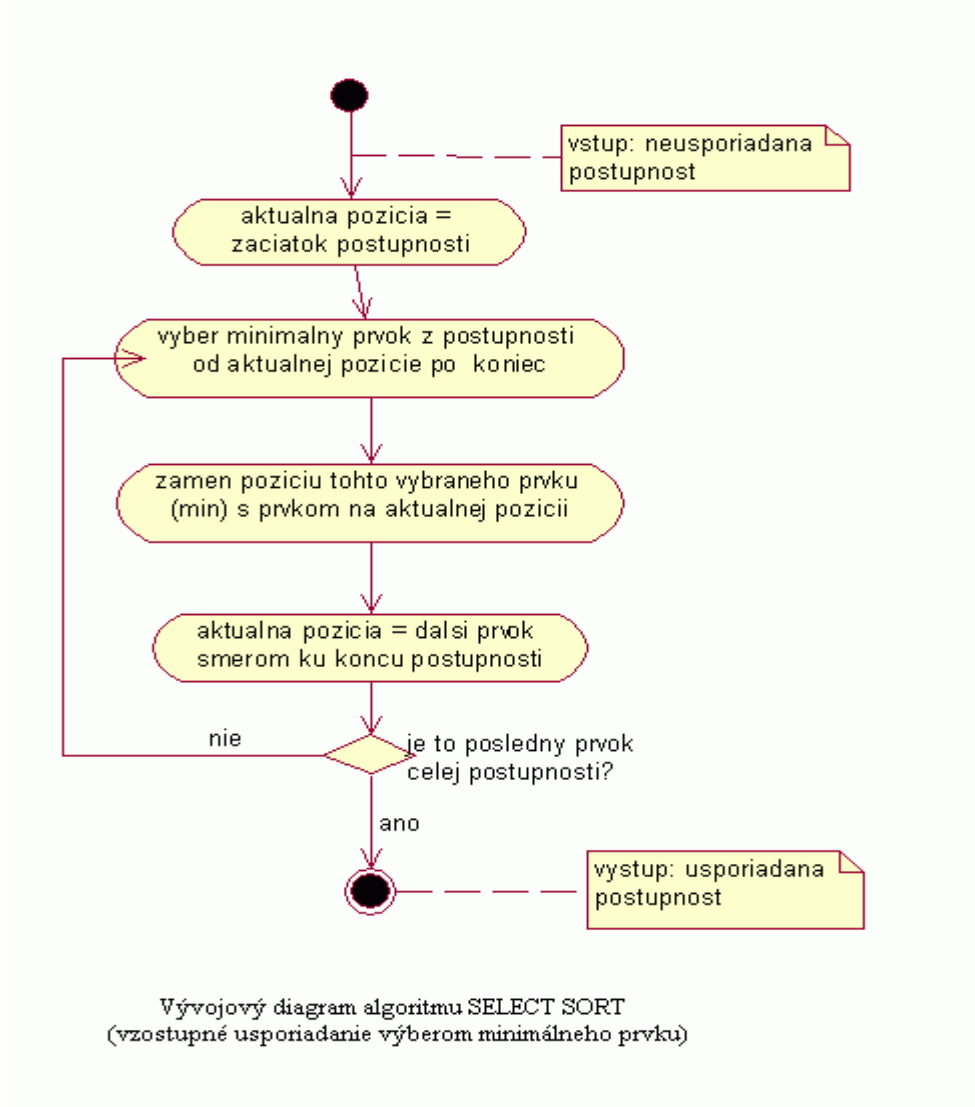
usporadúvania: vkladaním, výmenou, výberom, Shellovo, rýchle, zlučovaním, distributívne, radixové.

Helpful pages:
http://www.fiit.stuba.sk/~pospichal/soltis/uvod.htm
http://www.fiit.stuba.sk/~pospichal/molnar/system/index.html
http://www.fiit.stuba.sk/~pospichal/halanova/implementacia%20systemu/algoritmy.htm
(dost detailne info o jednotlivych triedeniach, + statistiky, zlozitosti, vizualne animacie)
http://www.developerfusion.co.uk/show/3824/ (A guide to sorting)
http://student.fiit.stuba.sk/~sobolic04/Triedenie.swf (pekna flashova animacia)

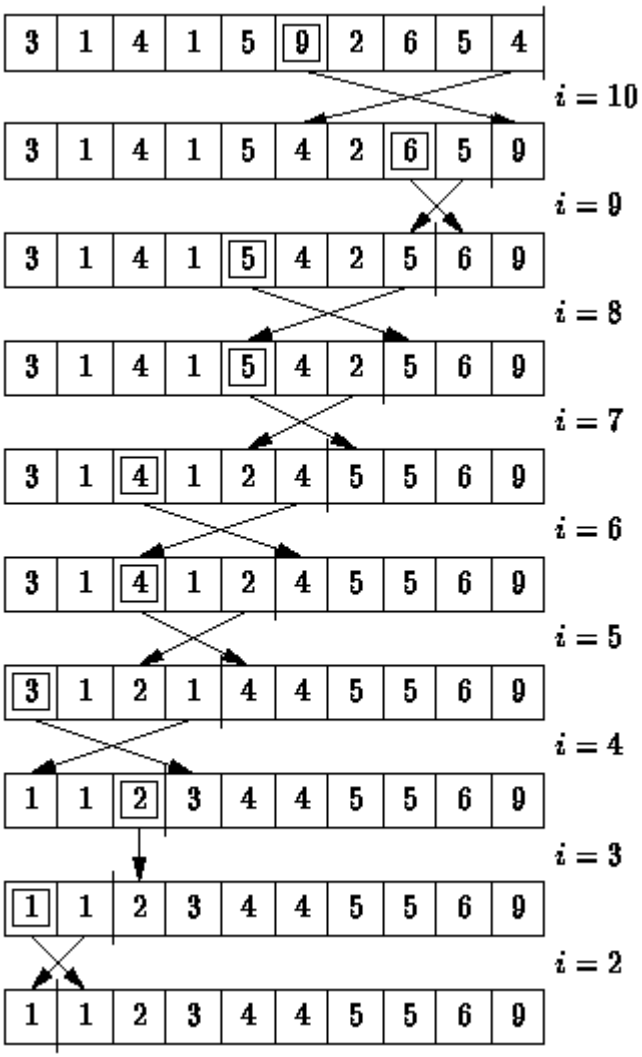
Select sort – Usporiadanie výberom

Metóda *Select sort* je algoritmus určený na usporadúvanie postupnosti. Je veľmi jednoduchý na implementáciu. *Select sort* patrí medzi pomalšie algoritmy pracujúce s operačnou zložitosťou $O(n^2)$.
Pri tejto metóde dochádza k porovnávaniu prvkov, preto *Select sort* patrí medzi komparačné algoritmy.
Algoritmus *Select sort* je univerzálny a je možné pomocou neho zoraďovať celé i reálne čísla a aj reťazce.

V podstate je tento algoritmus velmi jednoduchý. Prechádza postupnosťou a hladá najmenší (resp. najväčší) prvok a potom ho presunie na začiatok (resp. koniec) postupnosti. Toto sa opakuje až kým nebolo potrebné premiestniť ani jeden prvok.



Obr 2 . 16 Vývojový diagram algoritmu Selectionsort



Obr 2 . 17 Postup pri usporadúvaní algoritmom Selectionsort

Nasleduje funkcia v jazyku C, ktorá implementuje algoritmus *Select sort*. Uvedená funkcia vstupné pole celých čísel usporiada vzostupne.

```
//******SELECT SORT*******
//vstup: pole celych cisel urcene na usporiadanie, dlzka zadaneho pola
//ucel: vzostupne usporiadanie zadaneho pola
//algoritmus: Select sort vyber minimalneho prvku, vymena s prvym prvkom pola => vzostupne usporiadanie
//vystup: (void)
void int_select_vzostup_min(int *pole_int, int dlzka)
{
        int presuny = 0;
        int porovnania = 0;

        int min;                //minimalny prvok prvok
        int min_i = 0; //index minimalneho prvku
        int ptr = 0;            //ptr = aktualna pozicia
                                //zaciatok usporaduvaneho pola
        int i;                  //i = prehladavana pozicia
        int pom;                //pomocna premenna pri presuvani prvkov

        //pokial je aktualna pozicia(=ptr) mensia ako zadana dlzka pola
        while(ptr < dlzka){
                //32768 = maximum pre cele cisla (int) = pociatocna hodnota min pvrku
                min = 32768;
                //hladanie minima od aktualnej pozicie(=ptr) po koniec pola
                for(i = ptr; i < dlzka; i++){
                        porovnania++;
```

```
                        if ( pole_int[i] <= min ){
                                min = pole_int[i];
                                min_i = i;
                        }
                }

                //vymen prvok na aktualnej pozicii(=ptr) s minimalnym prvkom min
                if(ptr != min_i){
                        presuny = presuny+3;
                        pom = pole_int[min_i];
                        pole_int[min_i] = pole_int[ptr];
                        pole_int[ptr] = pom;
                }
                //aktualna pozicia(=ptr) = dalsi prvok pola
                ptr++;
        }

        //vypis konecnych hodnot porovnani a presunov
        printf("Pocet porovnani: %d\n", porovnania);
        printf("Pocet presunov: %d\n", presuny);

        return;
}
```

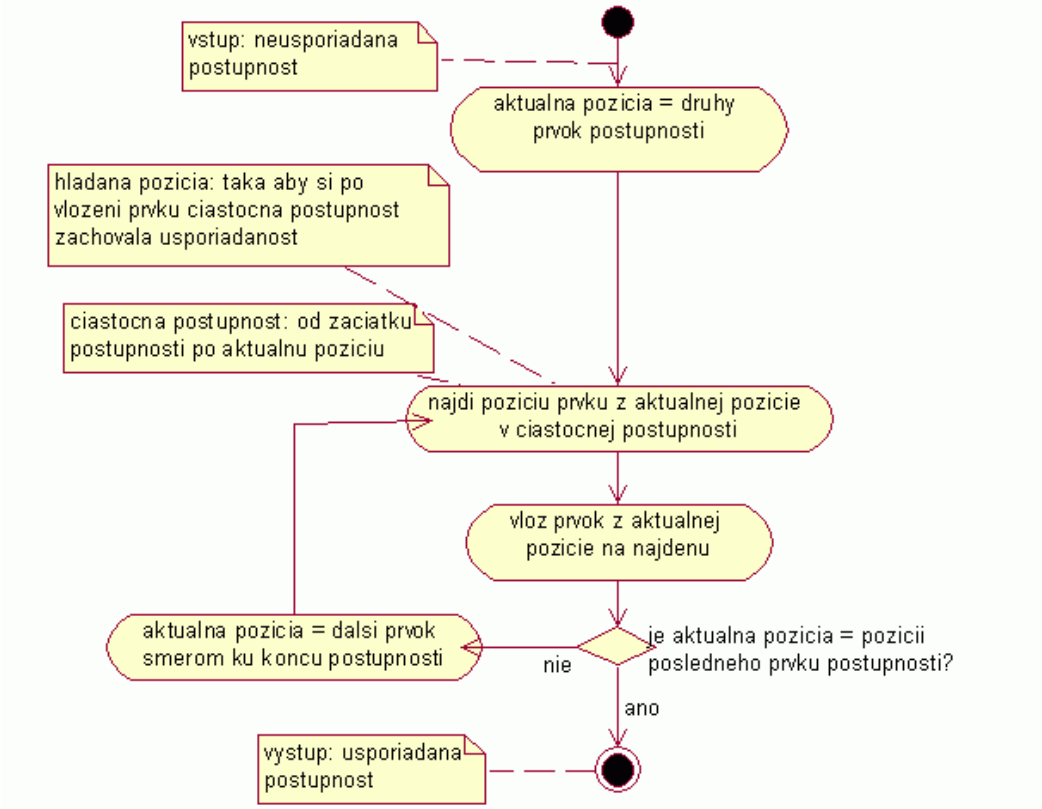
Insert sort – usporiadanie vkladaním

Základné vlastnosti

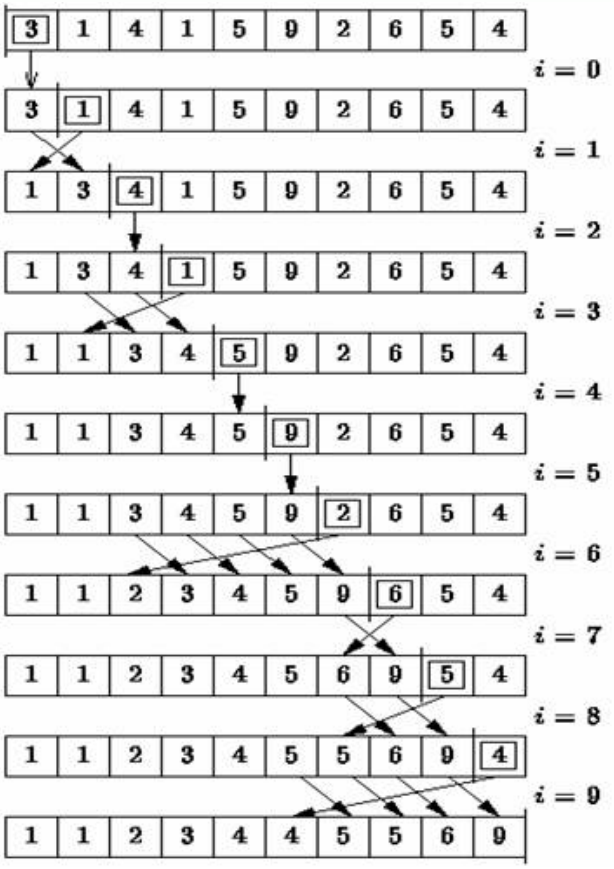
Metóda *Insert sort*, určená na usporadúvanie postupnosti prvkov **vkladaním**, patrí medzi jednoduché algoritmy usporadúvania.
Algoritmus *Insert sort* pracuje s operačnou zložitosťou $O(n^2)$.
Pri tejto metóde dochádza k porovnávaniu prvkov, preto patrí medzi komparačné algoritmy. Algoritmus *Insert sort* je univerzálny a je možné pomocou neho zoraďovať celé i reálne čísla a aj reťazce.

Popis

Pri zoraďovaní postupnosti o *n* prvkoch metóda zaručuje a zároveň využíva, že prvky *1 až i* (na začiatku $i = 2$) sú už zoradené.
Postupne prejdeme celú postupnosť od $i = 2$ *až po* $i < n$, každý prvok na pozícii *i* ukladáme do dočasnej premennej. Porovnávaním zistíme, na ktoré miesto v postupnosti s indexmi *1 až i-1* daný prvok patrí a vložíme ho tam. Takto je zaručené, že i po vložení prvku do postupnosti s indexmi *1 až i-1* zostáva táto postupnosť zoradená.
Usporiadanie je ukončené po zaradení posledného prvku.
Uvedený spôsob usporadúvania je možné realizovať pre vzostupné i zostupné usporiadanie. Konkrétne usporiadanie môžeme docieliť zmenou podmienok pri porovnávaní prvkov.



Vývojový diagram algoritmu INSERT SORT




```
//******INSERT SORT*******
//vstup: pole celych cisel urcene na usporiadanie, dlzka zadaneho pola
//ucel: vzostupne usporiadanie zadaneho pola
```

```
//algoritmus: Insert sort
//vystup: (void)
void int_insert_vzostup(int *pole_int, int dlzka)
{
	int presuny = 0;
	int porovnania = 0;

	int ptr = 1;		//ptr = aktualna pozicia
				//druhy prvok postupnosti
	int vkladany;  //vkladany prvok
	int pom_ptr;		//pomocny index
	int i;			//i = prehladavana pozicia

	//pokial je aktualna pozicia(=ptr) mensia ako zadana dlzka pola
	while(ptr < dlzka){
		//vyber vkladaneho prvku (z aktualnej pozicie=ptr)
		presuny++;
		vkladany = pole_int[ptr];

		//hladanie pozicie vkladaneho prvku v dosial usporiadanej casti pola,
		//tj. od zaciatku pola po aktualnu poziciu(=ptr)
		for(i = 0; i < ptr; i++){
			porovnania++;
			//pre zostupne usporiadanie bude podmienka:
			//if( vkladany > pole_int[i] )
			if( vkladany < pole_int[i] ){
				//vloz ho tam
				//presun prvkov, ktore budu v usporiadanom poli za
vkladanym
				pom_ptr = ptr;
				while(pom_ptr > i){
					presuny++;
					pole_int[pom_ptr] = pole_int[pom_ptr-1];
					pom_ptr--;
				}
				//vlozenie vkladaneho prvku na jeho miesto
				//i == ptr
				pole_int[i] = vkladany;
				presuny++;
				break;
			}
		}
		//if(i == ptr) netreba vkladany prvok vkladat,
		//pretoze je vacsi nez vsetky prvky v usporiadanej casti pola
		//aktualna pozicia(=ptr) = dalsi prvok pola
		ptr++;
	}

	//vypis konecnych hodnot porovnani a presunov
	printf("Pocet porovnani: %d\n", porovnania);
	printf("Pocet presunov: %d\n", presuny);

	return;
}
```

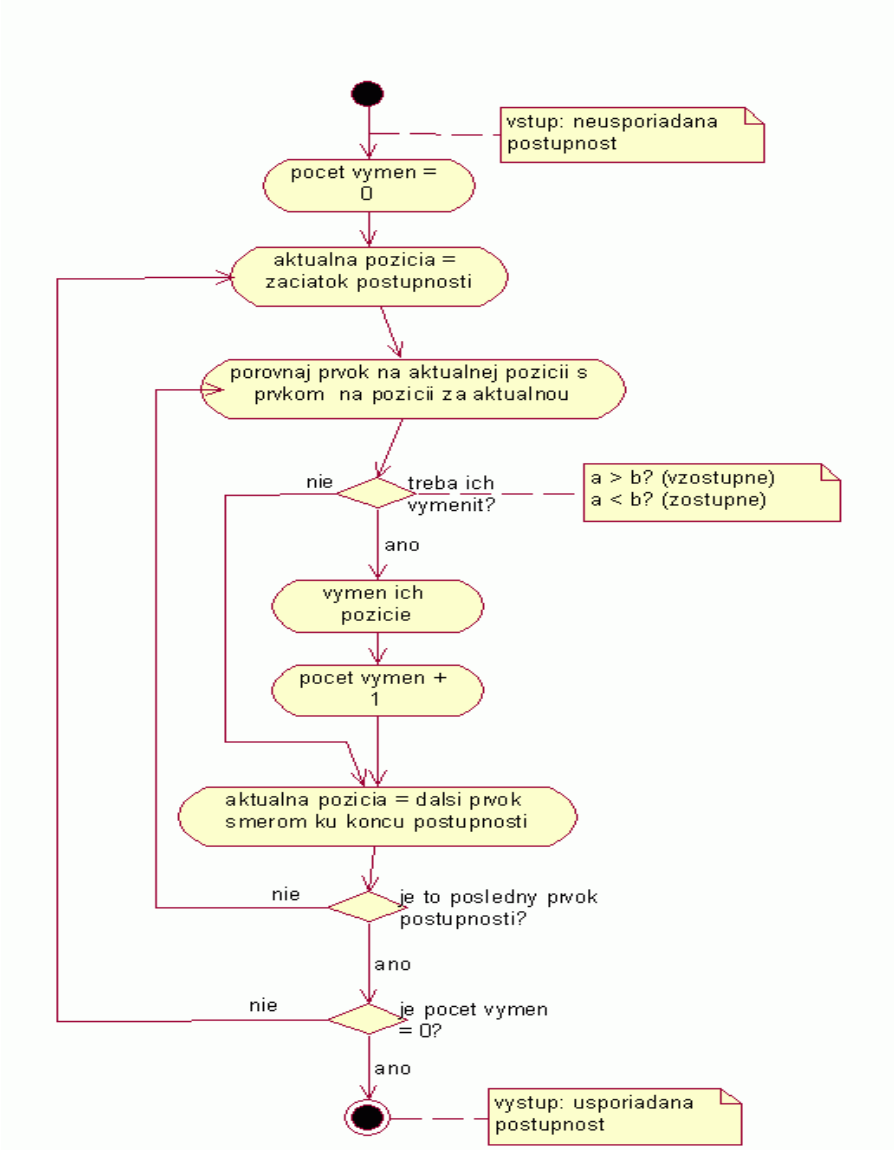
Bubble sort – usporiadanie výmenou

Základné vlastnosti

Bubble sort je algoritmus určený na usporadúvanie postupnosti prvkov. Operačná zložitosť algoritmu je $O(n^2)$. Je to najstarší, najjednoduchší, avšak i najpomalší algoritmus zo skupiny algoritmov s operačnou zložitosťou $O(n^2)$. *Bubble sort* je dnes prakticky nepoužívaný, avšak vďaka svojej jednoduchosti prežíva ako príklad jednoduchého usporadúvania vo výučbových procesoch.
Pri tejto metóde dochádza k porovnávaniu prvkov, čiže i *Bubble sort* patrí medzi komparačné algoritmy. Algoritmus *Bubble sort* je univerzálny a je možné pomocou neho zoraďovať celé i reálne čísla a aj reťazce.

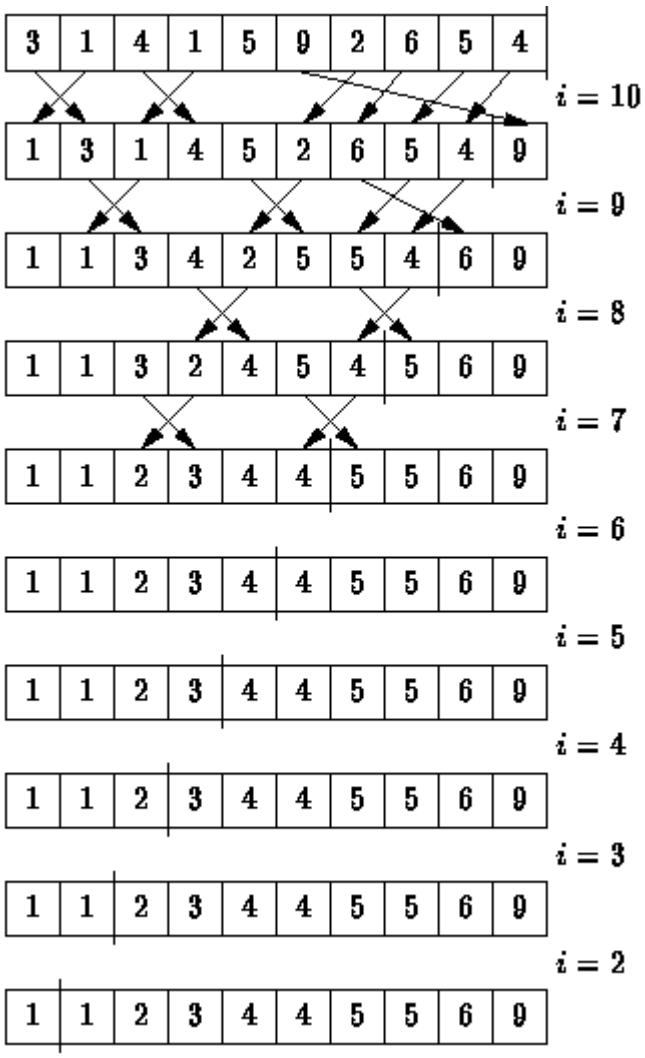
Popis

Vstupom pre algoritmus *Bubble sort* je neusporiadaná postupnosť. Algoritmus postupne prechádza celú postupnosť a porovnáva dva susedné prvky, v prípade potreby ich vymení. Pokiaľ v postupnosti dochádza k výmene prvkov, *Bubble sort* opätovne prechádza celú postupnosť a porovnáva všetky prvky usporadúvanej postupnosti. Ak pri prejdení celej postupnosti nenastane žiadna výmena, postupnosť je usporiadaná.
Bubble sort je možné použiť pre usporadúvania vzostupne i zostupne. Rozlíšenie týchto dvoch usporadúvaní sa deje pri porovnaní dvoch prvkov postupnosti. Kritérium porovnania si môžeme ľubovolne určiť podľa želaného výstupného usporiadania.



Vývojový diagram algoritmu BUBBLE SORT

Pozn: pocet vymen sa nuluje v kazdom cykle, a nie len raz, ako je to znazornene vo vyvojovom diagrame.



*//******BUBBLE SORT*******
//vstup: pole celych cisel urcene na usporiadanie, dlzka zadaneho pola
//ucel: vzostupne usporiadanie zadaneho pola
//algoritmus: Bubble sort
//vystup: (void)*

```
void int_bubble_vzostup(int *pole_int, int dlzka)
{
        int presuny = 0;
        int porovnania = 0;

        int pocet_vymen;        //pocet vymen pri jednom prechode celym zadanym polom
        int i;                  //i = aktualna pozicia
        int pom;                //pomocna premenna pri presuvani prvkov

        //aspon raz vykonaj
        do{
                pocet_vymen = 0;
                //pre cele pole, tj. od zaciatku az po dosiahnutie zadanej dlzky pola
                for(i =0; i < dlzka-1; i++){
                        porovnania++;
                        //pre zostupne usporiadanie bude podmienka:
                        //if(pole_int[i] < pole_int[i+1])
                        if( pole_int[i] > pole_int[i+1] ){
                                //vymen ich
                                presuny = presuny+3;
                                pom = pole_int[i];
                                pole_int[i] = pole_int[i+1];
                                pole_int[i+1] = pom;
                                //nastala vymena
                                pocet_vymen++;
                        }
                }
```

```
        //opakuj tento cyklus pokial nebude pocet_vymen = 0,
        //vtedy je zadane pole usporiadane
        } while(pocet_vymen != 0);

        //vypis konecnych hodnot porovnani a presunov
        printf("Pocet porovnani: %d\n", porovnania);
        printf("Pocet presunov: %d\n", presuny);

        return;
}
```

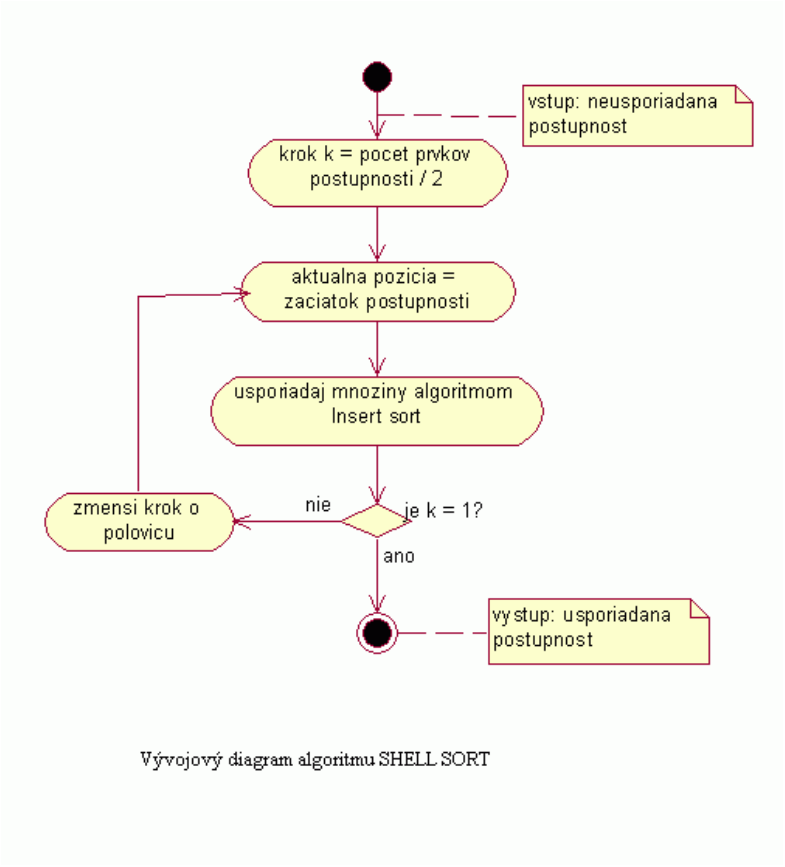
Shell sort – Shellovo usporiadanie

Základné vlastnosti

Shell sort je algoritmus určený na usporadúvanie postupnosti prvkov, pomenovaný podľa svojho autora Donalda Shella.
Zaraďujeme ho do skupiny algoritmov pracujúcich s operačnou zložitosťou $O(n^2)$. *Shell sort* je najefektívnejší avšak i najzložitejší algoritmus z tejto skupiny. [2]
Pri tejto metóde dochádza k porovnávaniu prvkov, preto *Shell sort* zaraďujeme medzi komparačné algoritmy. Algoritmus je univerzálny a je možné pomocou neho zoraďovať celé i reálne čísla a aj reťazce.

Popis

Pri usporadúvaní algoritmom *Shell sort* si na začiatku určíme nejaký krok. Spravidla sa počiatočný krok určí ako polovica počtu prvkov usporadúvanej postupnosti, teda krok $k=n/2$.
Podľa hodnoty k algoritmus *Shell sort* vytvorí množiny. Počet množín sa rovná hodnote k. Do množiny zaraďujeme tie prvky usporadúvanej postupnosti, ktoré sú na každej *k-tej* pozícii. Napríklad do prvej množiny pri hodnote kroku $k=3$ zaradíme prvky na pozíciách 1, 4, 7... , do druhej množiny by patrili prvky na pozíciách 2, 5, 8... a posledná tretia množina by obsahovala prvky na pozíciách 3, 6, 9... Takto rozdelíme prvky usporadúvanej postupnosti do k množín až po posledný prvok. Jednotlivé množiny sú spravidla usporadúvané algoritmom *Insert sort*. Po usporiadaní jednotlivých množín sú prvky v celkovej postupnosti usporiadané vzhľadom na pozície, z ktorých boli prvky do danej množiny zaradené. Celková postupnosť však nemusí byť usporiadaná. S krokom k zmenšeným o polovicu pôvodnej hodnoty kroku *Shell sort* opäť vytvorí množiny a pokračuje ďalej v usporadúvaní už spomenutým spôsobom. Algoritmus *Shell sort* končí po prejdení postupnosti s krokom $k=1$.
Výstupné usporiadanie je závislé od spôsobu usporiadania jednotlivých skupín algoritmom *Insert Sort*, ak použijeme vzostupné (zostupné) usporiadanie v rámci skupín, bude i výsledné usporiadanie vzostupné (zostupné).



Vývojový diagram algoritmu SHELL SORT

```
//******SHELL SORT*******
//vstup: pole celych cisel urcene na usporiadanie, dlzka zadaneho pola
//ucel: vzostupne usporiadanie zadaneho pola
//algoritmus: Shel sort
//vystup: (void)
void int_shell_vzostup(int *pole_int, int dlzka)
{
        int porovnania = 0;
        int presuny = 0;

        int krok = dlzka;
        int ptr;                //ptr = aktualna pozicia
        int ptr_zac;            //prt_zac = pociatocna aktualna pozicia
        int i;                  //i = prehladavana pozicia
        int pom_ptr;            //pomocny index
        int vkladany;   //vkladany prvok
        int poc_skup;   //pocet skupin

        //aspon raz vykonaj
        do{
                //zmensenie kroku o polovicu
                krok = krok/2;
                poc_skup = krok;
                ptr_zac = 0;

                while(poc_skup > 0){    //pre vsetky skupiny
                        poc_skup--;
                        ptr = ptr_zac;

                        //pre kazdu skupinu
                        //pokial je aktualna pozicia(=ptr)+krok mensia ako zadana dlzka
pola
                        while(ptr+krok < dlzka){
                                ptr = ptr+krok;
                                //vyber vkladaneho prvku z aktualnej pozicie(=ptr)
                                presuny++;
                                vkladany = pole_int[ptr];               //insert sort
                                //hladanie pozicie vkladaneho prvku v dosial usporiadanej
casti skupiny,
```

```
                                //tj. od zaciatku skupiny po aktualnu poziciu(=ptr) v
skupine
                                for(i = ptr_zac; i < ptr; ){
                                        porovnania++;
                                        //pre zostupne usporiadanie bude
                                        //nasledujuca podmienka: if( vkladany > pole_int[i]
)
                                        if( vkladany < pole_int[i] ){
                                                //vloz ho tam
                                                pom_ptr = ptr;
                                                //presun prvkov( budu v usporiadanej
skupine za vkladanym)
                                                while(pom_ptr > i){
                                                        presuny++;
                                                        pole_int[pom_ptr] =
pole_int[pom_ptr-krok];
                                                        pom_ptr = pom_ptr-krok;
                                                }
                                                //vlozenie vkladaneho prvku na jeho miesto
                                                //i == ptr
                                                presuny++;
                                                pole_int[i] = vkladany;
                                                break;
                                        }
                                        //prehladavana pozicia sa zvacsi o krok,
                                        //tzn. dalsi prvok danej skupiny
                                        i = i+krok;
                                }
                                //if(i == ptr) netreba vkladany vkladat, pretoze je
                                //vacsi nez vsetky prvky v usporiadanej casti skupiny
                        }
                        //zaciatocna aktualna pozicia(=ptr_zac) = dalsi prvok pola,
                        //tj. prvy prvok dalsej skupiny
                        ptr_zac++;
                }
        // opakuj tento cyklus pokial nie je krok=1, vtedy je zadane pole usporiadane
        } while(krok != 1);

        //vypis konecnych hodnot porovnani a presunov
        printf("Pocet porovnani: %d\n", porovnania);
        printf("Pocet presunov: %d\n", presuny);

        return;
}
```

Merge sort – usporiadanie výmenou

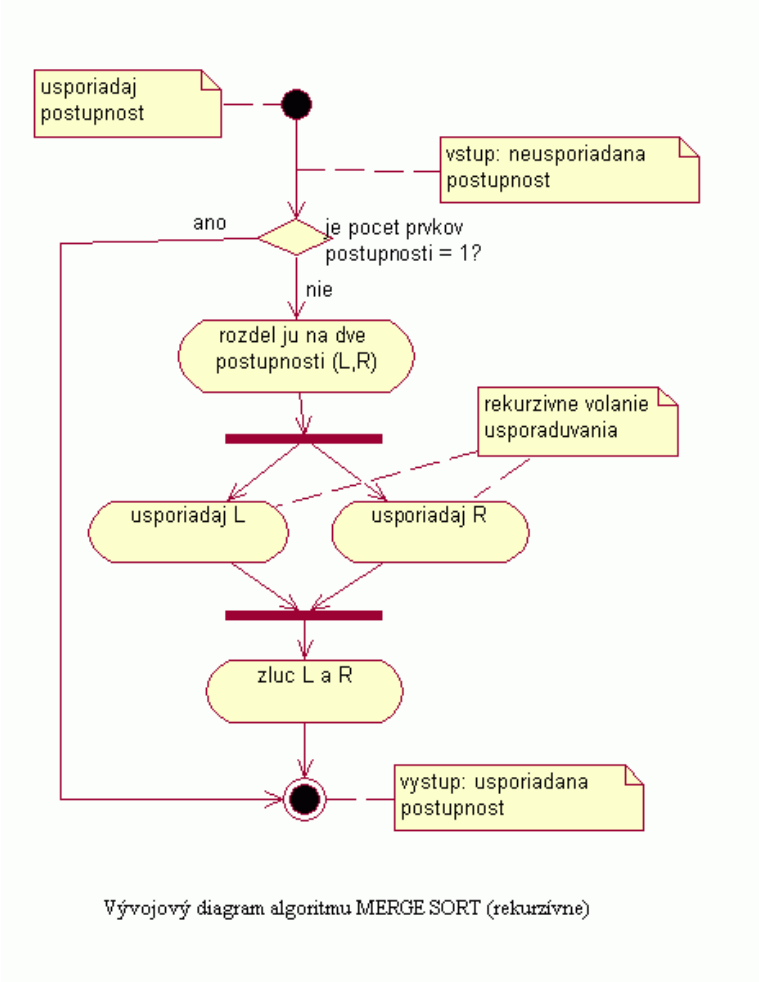
Základné vlastnosti

Merge sort je algoritmus určený na usporadúvanie postupnosti prvkov zlučovaním.
Operačná zložitosť algoritmu je O(n*log2n).
Pri tejto metóde dochádza k porovnávaniu prvkov, preto *Merge sort* patrí medzi komparačné algoritmy.
Algoritmus *Merge sort* je univerzálny a je možné pomocou neho zoraďovať celé i reálne čísla a aj reťazce.
Merge sort je rekurzívny algoritmus, avšak je možné vytvoriť i nerekurzívnu formu so zachovaním základnej filozofie algoritmu.

Popis

Algoritmus *Merge Sort* je založený na zlučovaní dvoch už usporiadaných postupností. Vstupnú neusporiadanú postupnosť rozdelíme na dve, po usporiadaní týchto dvoch postupností ich zlúčime a dostaneme výslednú usporiadanú postupnosť.
Vstupom pre algoritmus sú dve usporiadané polia *a* a *b* (postupnosti) a výstupom usporiadané pole *c*, ktoré vzniklo zlúčením (merge) oboch vstupných polí. Pri vykonávaní algoritmu sú použité tri ukazovatele (*a_ptr, b_ptr, c_ptr*), a to ukazovatele na aktuálnu pozíciu v každom poli. Porovnáme prvky *a[a_ptr]* a *b[b_ptr]* ,

menší z nich je umiestnený do postupnosti *c* na pozíciu *c[c_ptr]* . Po presunutí sú dané dva ukazovatele (*a_ptr* alebo *b_ptr* a *c_ptr*) posunuté o jednu pozíciu. Akonáhle je jedna zo vstupných postupností prázdna (celá presunutá do výstupnej postupnosti), je zvyšok druhej postupnosti presunutý do výstupnej bezo zmeny poradia prvkov.
Uvedený spôsob zoraďuje postupnosť vzostupne. Zmenou podmienky pri porovnávaní prvkov docielime zostupné usporiadanie.



Vývojový diagram algoritmu MERGE SORT (rekurzívne)

```
//******MERGE SORT*******
//vstup: pole celych cisel urcene na usporiadanie, dlzka zadaneho pola
//ucel: vzostupne usporiadanie zadaneho pola
//algoritmus: Merge sort, rekurzivna podoba algoritmu
//vystup: (void)
void int_merge_vzostup(int *pole, int dlzka)
{
        //ak je dlzka rozna od 1, je potrebne zadane pole usporiadat
        if(dlzka > 1){
                //dve casti povodnej postupnosti
                int *lava;
                int *prava;
                int dlzka_lava, dlzka_prava;

                //rozdelenie zadaneho pola na dve polivice
                dlzka_lava = dlzka/2;
                dlzka_prava = dlzka-dlzka_lava;

                lava = (int *) malloc(dlzka_lava*sizeof(int));
                prava = (int *) malloc(dlzka_prava*sizeof(int));

                for(int i = 0; i < dlzka_lava; i++){
                lava[i] = pole[i];
                        prava[i] = pole[i+dlzka_lava];
                }
                //ak v povodnej casti postupnosti este nieco zostalo
                //dame to do pravej postupnosti
```

```
                if(dlzka_lava != dlzka_prava){
                //if(i+dlzka_lava < dlzka-1){
                        for(i = dlzka_lava; i < dlzka_prava; i++){
                                //presuny++;
                                prava[i] = pole[i+dlzka_lava];
                        }
                }

                //rekurzivne volanie
                rek_volania = rek_volania+2;
                int_merge_vzostup(lava, dlzka_lava);
                int_merge_vzostup(prava, dlzka_prava);

                //zlucenie == MERGE
                int i_lava = 0, i_prava = 0;   //aktualne pozicie v zlucovanych poliach
                //pre cele pole, ktore vznika zlucovanim
                //i = aktualna pozicia vo vystupnom poli
                for(i = 0; i < dlzka; i++){
                        //ak v lavej zlucovanej casti uz nie su ziadne prvky
                        if(i_lava > dlzka_lava-1){
                                //skopiruj vsetko z pravej casti do vysledneho pola
                                presuny++;
                                pole[i] = prava[i_prava];
                                i_prava++;
                        }else{
                                //ak v pravej zlucovanej casti nie su uz ziadne prvky
                                if(i_prava > dlzka_prava-1){
                                        //skopiruj vsetko z lavej casti do vysledneho pola
                                        presuny++;
                                        pole[i] = lava[i_lava];
                                        i_lava++;
                                }else{
                                        //inak treba porovnat prve prvky v lavej a pravej
casti
                                        porovnania++;
                                        if(lava[i_lava] < prava[i_prava]){
                                                //presun prvku z lavej casti do vysledneho
pola
                                                presuny++;
                                                pole[i] = lava[i_lava];
                                                i_lava++;
                                        }else {
                                                //presun prvku z pravej casti do vysledneho
pola
                                                presuny++;
                                                pole[i] = prava[i_prava];
                                                i_prava++;
                                        }
                                }
                        }
                }
        }

        return;
}
```

Quick sort – rýchle usporadúvanie

Základné vlastnosti

Metóda *Quick sort* vynájdená pánom C.A.R. Hoare patrí medzi efektívne algoritmy usporadúvania. Už jej názov napovedá, že ide o rýchly algoritmus. *Quick sort* zaraďujeme medzi metódy pracujúce s operačnou zložitosťou O(n*log2n).
Keďže pri tejto metóde dochádza k porovnávaniu prvkov, *Quick sort* patrí medzi komparačné algoritmy.
Algoritmus *Quick sort* je univerzálny a je možné pomocou neho zoraďovať celé i reálne čísla a aj reťazce.

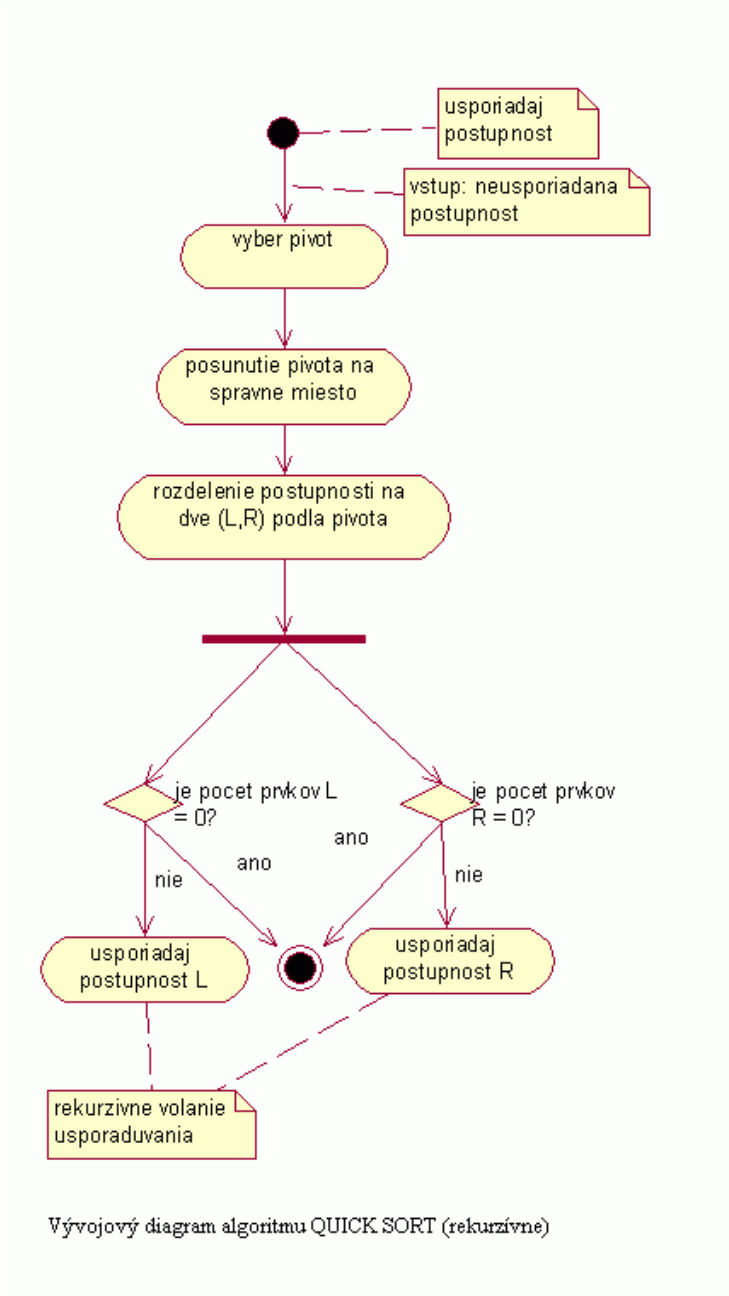
Quick sort pracuje rekurzívne, avšak je možné tento algoritmus prepísať i do formy nerekurzívnej, ako uvidíme v nasledujúcej časti.

Popis

Myšlienka metódy spočíva v postupnom rozdeľovaní poľa na dve menšie polia a v ich následnom usporiadaní. Polia sa rozdeľujú podľa náhodne zvoleného pivota. Pivot je definovaný ako prvok postupnosti, od ktorého prvky naľavo sú menšie než pivot a napravo väčšie než pivot. Je to vlastne prvok, ktorý je aj bez triedenia na svojom mieste.
Matematicky:
Číslo *T* v postupnosti *x1, x2,...,xr=T,...,xn* nazývame pivot, ak pre všetky *i* platí:
pre *i = 1,...,(r-1) xi < =T*
a pre *i = (r+1), ..., n xi >= T*

Rekurzívne riešenie algoritmu :
Vstupom pre algoritmus je neusporiadané pole a dva indexy poľa. Vstupné indexy *lavy* a *pravy* určujú, medzi ktorými indexmi sa bude postupnosť usporadúvať.
V zadanej postupnosti si môžeme za pivota zvoliť ľubovolný prvok. Zvoľme prvý prvok spracúvanej časti postupnosti. Postupne prechádzame celú postupnosť a porovnávame všetky prvky s pivotom. Ak na pravej strane (presúvame sa z konca postupnosti smerom k začiatku) nájdeme prvok menší než pivot, tento prvok presunieme do ľavej (aktuálne najľavejšej) časti spracúvanej postupnosti. Postupne sa indexom približujeme k stredu postupnosti. Potom prechádzame ľavú časť (v postupnosti od začiatku smerom ku stredu). Ak v ľavej časti nájdeme prvok väčší než pivot, presunieme ho do pravej časti. Ľavým indexom sa tiež približujeme k stredu postupnosti. Za stred sa považuje miesto, kde treba zaradiť aktuálny pivot, aby sa tento nachádzal na svojom mieste. Po prejdení ľavej aj pravej časti, skontrolovaní a presunutí potrebných prvkov dáme pivot medzi tieto postupnosti, tzn., že tento prvok sa nachádza na svojom mieste (vľavo sú prvky menšie, vpravo väčšie). Rovnakým spôsobom prehľadáme a upravíme ľavú podpostupnosť a pravú podpostupnosť zvlášť, pričom pôvodný pivot si zachováva pozíciu, na ktorú bol presunutý (medzi podpostupnosti). Rekurzívnym volaním funkcie tak zabezpečíme upravenie celej postupnosti rozkladom na podpostupnosti. Môže nastať prípad, že po umiestnení pivota na jeho miesto sa tento nachádza na začiatku, resp. konci spracovávanej postupnosti, v takomto prípade je ľavá, resp. pravá podpostupnosť nulová. Vtedy je potrebné usporiadať iba druhú nenulovú podpostupnosť spracovávanej postupnosti. Rekurzívne volanie pre nulovú postupnosť sa teda nevykonáva, čo predstavuje ukončovaciu podmienku tohto algoritmu pre danú časť postupnosti.

Nerekurzívne riešenie:
Pred volaním funkcie usporadúvania je potrebné indexy *lavy* a *pravy* ukladať do zásobníka. Potrebu ukladania indexov do zásobníka nemožno považovať za nedostatok, pretože pri rekurzívnom volaní sú do zásobníka ukladané hodnoty všetkých lokálnych premenných, čo kladie na systém väčšie nároky. Pri nerekurzívnom riešení beží algoritmus v cykle a usporadúva čiastkové postupnosti medzi danými indexmi, pokým nie je zásobník prázdny, tzn. pokým sa v zásobníku nachádzajú hodnoty indexov *lavy* a *pravy.*



Vývojový diagram algoritmu QUICK SORT (rekurzívne)

//prve volanie funkcie
int_q_sort_vzostup(pole_int, 0, dlzka-1);

*//******QUICK SORT*******
//vstup: pole celych cisel urcene na usporiadanie, lavy a pravy index, medzi ktorymi bude pole usporiadane (pri prvom volani: lavy = 0 a pravy = dlzka-1)
//ucel: vzostupne usporiadanie zadaneho pola
//algoritmus: Quick sort, rekurzivna podoba algoritmu
//vystup: (void)*

```
void int_q_sort_vzostup(int *pole, int lavy, int pravy)
{
        int pivot_index, zac_lavy, zac_pravy; //indexy
        int pivot;                            //pivot

        //v zac_lavy, zac_pravy si uchovame pociatocnu hodnotu lavy a pravy
        zac_lavy = lavy;
        zac_pravy = pravy;

        //zvolime si pivot
        //za pivota vyberame prvy prvok v "lavej" casti pola,tj podla "laveho" indexu
        presuny++;
        pivot = pole[lavy];
```

```
	while (lavy < pravy){
		//testujeme ci je prvok v "pravej" casti pola vacsi ako pivot
		//pre zostupne usporiadanie bude podmienka:
		//while( (pole[pravy] <= pivot) && (lavy < pravy) )
		while( (pole[pravy] >= pivot) && (lavy < pravy) ){
			porovnania++;
			pravy--;
		}
		if(lavy < pravy)
			porovnania++;

		//ak najdeme prvok mensi ako pivot(v pravej casti) a ich indexy sa
nezhoduju,
		//tj su to dva rozne prvky, tak na miesto prvku, ktory je na pozicii lavy
		//dame tento mensi prvok z "pravej" casti
		//pricom pivot(hodnota)ostava nezmeneny
		if (lavy != pravy){
			presuny++;
			pole[lavy] = pole[pravy];
			lavy++;
			//posunieme sa v lavej casti a prave skopirovany prvok z pravej
casti sa
			//pri porovnavani lavej casti pola v tomto cykle uz neporovnava
			//pretoze uz vieme ze je mensi nez pivot
		}

		//testujeme, ci su prvky v "lavej" casti pola mensie ako pivot
		//pre zostupne usporiadanie bude podmienka:
		//while( (pole[lavy] <= pivot) && (lavy < pravy) )
		while( (pole[lavy] <= pivot) && (lavy < pravy) ){
			porovnania++;
			lavy++;
		}
		if(lavy < pravy)
			porovnania++;

		//ak najde prvok vacsi ako pivot a tento sa nachadza v lavej casti
		//skopiruje tento prvok do pravej casti na miesto, z ktoreho bol predtym
kopirovany
		//mensi prvok z pravej casti do lavej
		//ak predtym nebol najdeny ziaden prvok v pravej casti mensi ako pivot
		//tak indexy lavy a pravy sa rovnaju,
		//tj. vstupne podmienky(lavy != pravy, lavy < pravy) su nesplnene
		if (lavy != pravy){
			presuny++;
			pole[pravy] = pole[lavy];
			pravy--;
		}
	}

	//nakoniec sa pivot skopiruje na miesto ktore oddeluje pravu a lavu cast pola
	//toto miesto je teraz dane indexom lavy
	presuny++;
	pole[lavy] = pivot;
	pivot_index = lavy;

	//do lavy a pravy dame hodnoty, ktore mali tieto indexy na zaciatku volania
funkcie
	lavy = zac_lavy;
	pravy = zac_pravy;
	//volanie funkcie q_sort pre lavu cast prave spracovavaneho pola
	if (lavy < pivot_index){
		rek_volania++;
		int_q_sort_vzostup(pole, lavy, pivot_index-1);
	}
	//volanie funkcie q_sort pre pravu cast prave spracovavaneho pola
	if (pravy > pivot_index){
		rek_volania++;
		int_q_sort_vzostup(pole, pivot_index+1, pravy);
```

```
        }

        return;
}
```

nerekurzívna verzia so zásobníkom:

Solution #1: Modifed quicksort, non-recursive.

In this implementation we will make a number of changes. First we will see how we can use a stack to remove recursion. Also, we will choose the split- point more carefully, as a median of the first, the middle and the last elements. This will make the worst case behaviour more difficult to achieve. Also, we will try a different splitting strategy. We will also look at the sizes of the sub-lists and stack only one, smaller sub-list. This will make the worst case stack space come down to O(lg(n)) from O(n). Also, to sort the sub-lists of size less than 10, we will use insertion sort. This will also speed up the process, since for small lists, insertion sort is better than quicksort.

Let us look at the program:

```
#include <stdio.h>

#define MAXELT          100
#define INFINITY        32760                   //numbers in list should not exceed
                                                //this. change the value to suit your
                                                //needs
#define SMALLSIZE       10                      //not less than 3
#define STACKSIZE       100                     //should be ceiling(lg(MAXSIZE)+1)

int list[MAXELT+1];                             //one extra, to hold INFINITY

struct {                                        //stack element.
        int a,b;
} stack[STACKSIZE];

int top=-1;                                     //initialise stack

void main()                                     //overhead!
{
        int i=-1,j,n;
        char t[10];
        void quicksort(int);

        do {
                if (i!=-1)
                        list[i++]=n;
                else
                        i++;
                printf("\nEnter the numbers <End by #>");
                fflush(stdin);
                scanf("%[^\n]",t);
                if (sscanf(t,"%d",&n)<1)
                        break;
        } while (1);

        quicksort(i-1);

        printf("The list obtained is ");

        for (j=0;j<i;j++)
                printf("\n %d",list[j]);
}

void interchange(int *x,int *y)                 //swap
```

```
{
        int temp;

        temp=*x;
        *x=*y;
        *y=temp;
}

void split(int first,int last,int *splitpoint)
{
        int x,i,j,s,g;

        //here, atleast three elements are needed
        if (list[first]<list[(first+last)/2]) {  //find median
                s=first;
                g=(first+last)/2;
        }
        else {
                g=first;
                s=(first+last)/2;
        }
        if (list[last]<=list[s])
                x=s;
        else if (list[last]<=list[g])
                x=last;
        else
                x=g;
        interchange(&list[x],&list[first]);      //swap the split-point element
                                                 //with the first
        x=list[first];
        i=first;                                 //initialise
        j=last+1;
        while (i<j) {
                do {                             //find j
                        j--;
                } while (list[j]>x);
                do {
                        i++;                     //find i
                } while (list[i]<x);
                interchange(&list[i],&list[j]);  //swap
        }
        interchange(&list[i],&list[j]);          //undo the extra swap
        interchange(&list[first],&list[j]);      //bring the split-point
                                                 //element to the first
        *splitpoint=j;
}

void push(int a,int b)                           //push
{
        top++;
        stack[top].a=a;
        stack[top].b=b;
}

void pop(int *a,int *b)                          //pop
{
        *a=stack[top].a;
        *b=stack[top].b;
        top--;
}

void insertion_sort(int first,int last)
{
        int i,j,c;

        for (i=first;i<=last;i++) {
                j=list[i];
                c=i;
                while ((list[c-1]>j)&&(c>first)) {
```

```
                        list[c]=list[c-1];
                        c--;
                }
                list[c]=j;
        }
}

void quicksort(int n)
{
        int first,last,splitpoint;

        push(0,n);
        while (top!=-1) {
                pop(&first,&last);
                for (;;) {
                        if (last-first>SMALLSIZE) {
                                                //find the larger sub-list
                                split(first,last,&splitpoint);
                                                //push the smaller list
                                if (last-splitpoint<splitpoint-first) {
                                        push(first,splitpoint-1);
                                        first=splitpoint+1;
                                }
                                else {
                                        push(splitpoint+1,last);
                                        last=splitpoint-1;
                                }
                        }
                        else {                  //sort the smaller sub-lists
                                                //through insertion sort
                                insertion_sort(first,last);
                                break;
                        }
                }
        }                                       //iterate for larger list
}
//end of solution 2
```

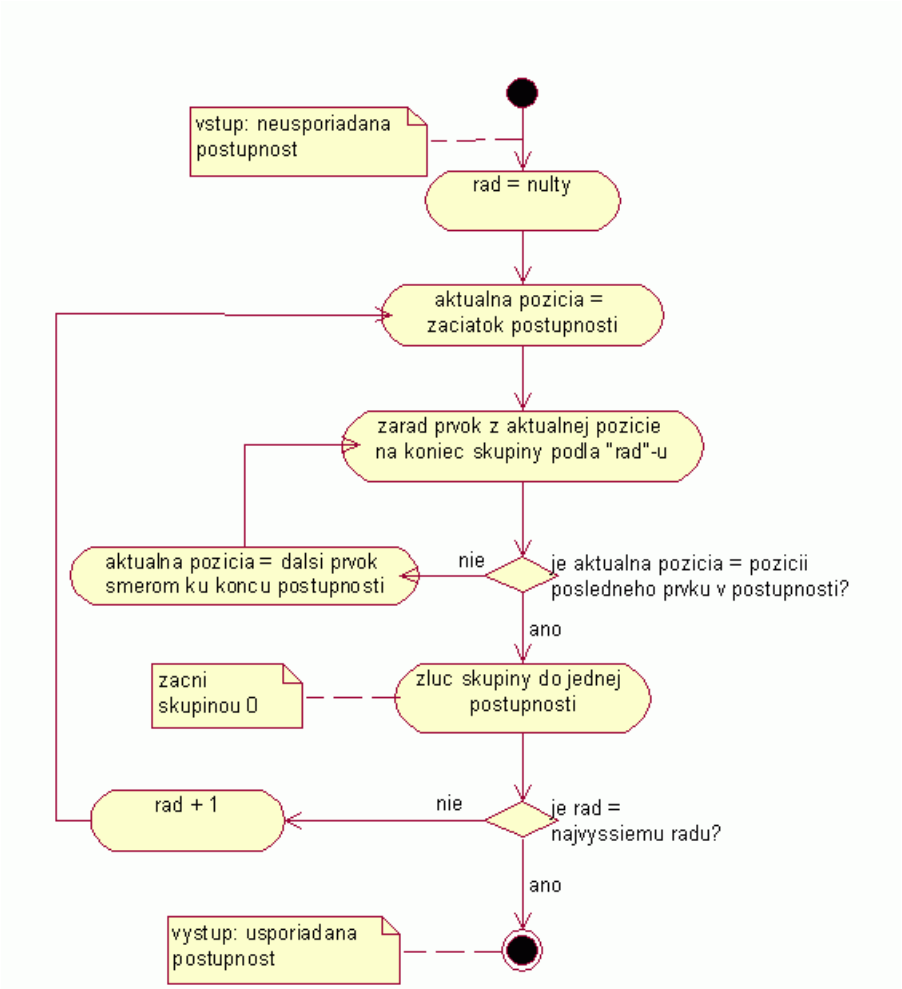
Radix sort

Základné vlastnosti

Radix sort bol používaný najmä pri usporadúvaní diernych štítkov, dnes už je takmer nepoužívaný. [8]
Metóda *Radix sort* je určená na usporadúvanie postupnosti prvkov. *Radix sort* patrí medzi najrýchlejšie algoritmy usporadúvania, avšak vo všeobecnosti nie je rýchlejší než *Quick sort. Radix sort* je najčastejšie používaný pri usporadúvaní desiatkových čísel, avšak môžeme použiť ľubovolný základ pre usporadúvané čísla. [3]
Operačná zložitosť algoritmu je O(n*log2n).
Radix sort je nekomparačný algoritmus.

Popis

V algoritme *Radix sort* jednotlivé členy vstupnej neusporiadanej postupnosti triedime do skupín. Skupina predstavuje postupnosť prvkov, ktoré majú rovnakú hodnotu *i-teho* rádu, kde *i+1* je krok vykonávania algoritmu. V prvom kroku je príslušnosť do skupiny určená najnižším, teda nultým rádom radených prvkov. Postupne radíme prvky do skupín podľa vyšších rádov až po posledný rád. Počet krokov algoritmu závisí od rádu prvku s najvyšším rádom. Vstupom pre prvý krok triedenia je usporadúvaná postupnosť, pre triedenia do skupín v ďalších krokoch je vstupom postupnosť, ktorá vznikne spojením čiastkových postupností z jednotlivých skupín v predchádzajúcom kroku, pričom spájanie začíname od nultej skupiny.
Uvedený spôsob usporiada prvky vstupnej postupnosti vzostupne, zmenou poradia spájania skupín do postupností, ktoré sú vstupom pre nasledujúce triedenie, môžeme docieliť zostupné usporiadanie.



Vývojový diagram algoritmu RADIX SORT (vzostupné usporiadanie)

```
//******RADIX SORT*******
//vstup: pole celych cisel urcene na usporiadanie, dlzka zadaneho pola
//ucel: vzostupne usporiadanie zadaneho pola
//algoritmus: Radix sort
//vystup: (void)
void int_radix_vzostup(int *pole, int dlzka)
{
        int i, j;
        //rad, podla ktoreho prave usporaduvame (rozdelujeme prvky do skupin)
        int rad = 1;
        int ptr;                //ptr = aktualna pozicia
        int rad_pom = 0;
        //rad_pom urcuje pocet prvkov usporaduvanej postupnosti,
        //ktore su nizsieho radu nez je rad, podla ktoreho sa aktualne radi do skupin
        //ked rad_pom == dlzka koncime

        //najhorsie riesenie
        //(skupiny = polia rovnakej dlzky ako usporaduvane pole, index do kazdeho pola)
        int *nulta_skup;
        nulta_skup = (int*) malloc(dlzka*sizeof(int));
        int nulta;
        int *prva_skup;
        prva_skup = (int*) malloc(dlzka*sizeof(int));
        int prva;
        int *druha_skup;
        druha_skup = (int*) malloc(dlzka*sizeof(int));
        int druha;
        int *tretia_skup;
        tretia_skup = (int*) malloc(dlzka*sizeof(int));
        int tretia;
```

```
        int *stvrta_skup;
        stvrta_skup = (int*) malloc(dlzka*sizeof(int));
        int stvrta;
        int *piata_skup;
        piata_skup = (int*) malloc(dlzka*sizeof(int));
        int piata;
        int *siesta_skup;
        siesta_skup = (int*) malloc(dlzka*sizeof(int));
        int siesta;
        int *siedma_skup;
        siedma_skup = (int*) malloc(dlzka*sizeof(int));
        int siedma;
        int *osma_skup;
        osma_skup = (int*) malloc(dlzka*sizeof(int));
        int osma;
        int *deviata_skup;
        deviata_skup = (int*) malloc(dlzka*sizeof(int));
        int deviata;

        //pokial este ostavaju prvy vyssieho radu nez je ten, podla ktoreho radime do
skupin
        while(rad_pom < dlzka){
                //atualna pozicia a pozicie v skupinach su na danych prvych prvku
                ptr = 0;
                nulta = 0;
                prva = 0;
                druha = 0;
                tretia = 0;
                stvrta = 0;
                piata = 0;
                siesta = 0;
                siedma = 0;
                osma = 0;
                deviata = 0;
                //pokial je aktualna pozicia(=ptr) mensia ako zadana dlzka pola
                while(ptr < dlzka){
                        //do premennej i dame hodnotu vyssich radov
                        //(nez aktualny rad) prvku na aktualnej pozicii
                        i = pole[ptr]/(rad*10);
                        //ak je i = 0, je aktualny rad najvyssim radom daneho prvku
                        if(i == 0){
                                rad_pom++;
                        }
                        //do premennej i dame hodnotu aktualneho radu
                        i = ( pole[ptr]%(rad*10) ) / rad;

                        //podla hodnoty tohto radu rozhodneme,
                        //do ktorej skupiny prvok na aktualnej pozicii zaradime
                        switch(i){
                        case 0:
                                nulta_skup[nulta] = pole[ptr];
                                nulta++;
                                break;
                        case 1:
                                prva_skup[prva] = pole[ptr];
                                prva++;
                                break;
                        case 2:
                                druha_skup[druha] = pole[ptr];
                                druha++;
                                break;
                        case 3:
                                tretia_skup[tretia] = pole[ptr];
                                tretia++;
                                break;
                        case 4:
                                stvrta_skup[stvrta] = pole[ptr];
                                stvrta++;
                                break;
```

```
                case 5:
                        piata_skup[piata] = pole[ptr];
                        piata++;
                        break;
                case 6:
                        siesta_skup[siesta] = pole[ptr];
                        siesta++;
                        break;
                case 7:
                        siedma_skup[siedma] = pole[ptr];
                        siedma++;
                        break;
                case 8:
                        osma_skup[osma] = pole[ptr];
                        osma++;
                        break;
                case 9:
                        deviata_skup[deviata] = pole[ptr];
                        deviata++;
                        break;
                }
                //aktualna pozicia(=ptr) = dalsi prvok pola
                ptr++;
        }
        //spojenie skupin do jedneho pola, zaciname od nultej
        for(j = 0; j < nulta; j++){
                pole[j] = nulta_skup[j];
        }
        int pom = nulta;
        for(j = 0; j < prva; j++){
                pole[j+pom] = prva_skup[j];
        }
        pom += prva;
        for(j = 0; j < druha; j++){
                pole[j+pom] = druha_skup[j];
        }
        pom += druha;
        for(j = 0; j < tretia; j++){
                pole[j+pom] = tretia_skup[j];
        }
        pom += tretia;
        for(j = 0; j < stvrta; j++){
                pole[j+pom] = stvrta_skup[j];
        }
        pom += stvrta;
        for(j = 0; j < piata; j++){
                pole[j+pom] = piata_skup[j];
        }
        pom += piata;
        for(j = 0; j < siesta; j++){
                pole[j+pom] = siesta_skup[j];
        }
        pom += siesta;
        for(j = 0; j < siedma; j++){
                pole[j+pom] = siedma_skup[j];
        }
        pom += siedma;
        for(j = 0; j < osma; j++){
                pole[j+pom] = osma_skup[j];
        }
        pom += osma;
        for(j = 0; j < deviata; j++){
                pole[j+pom] = deviata_skup[j];
        }
        //zvysenie radu
        rad*=10;

}
```

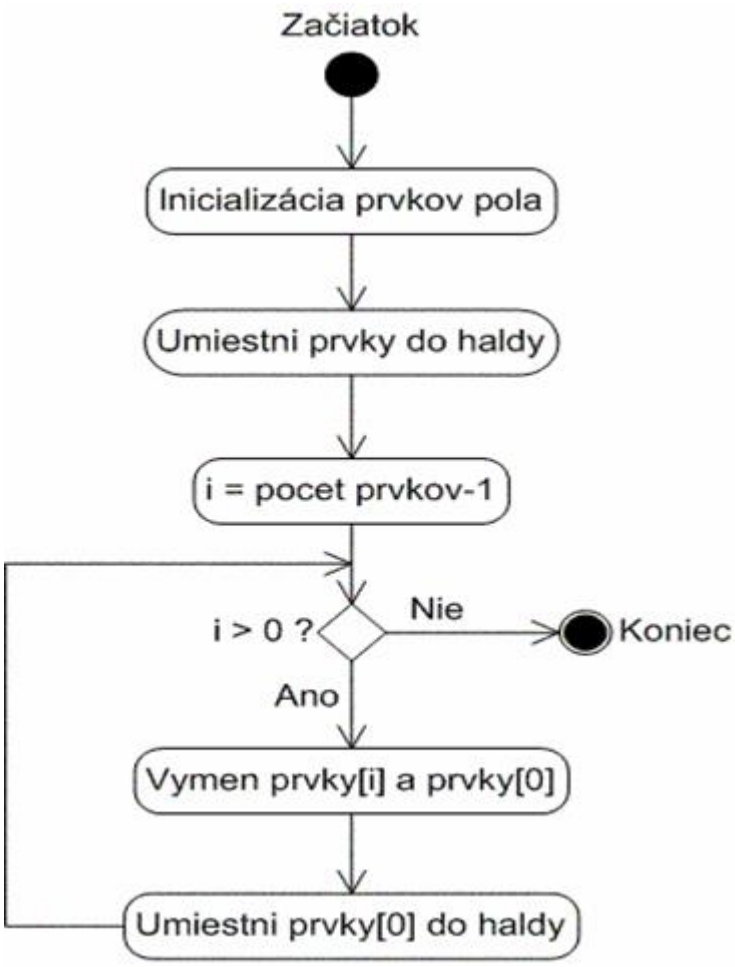
```
        return;
}
```

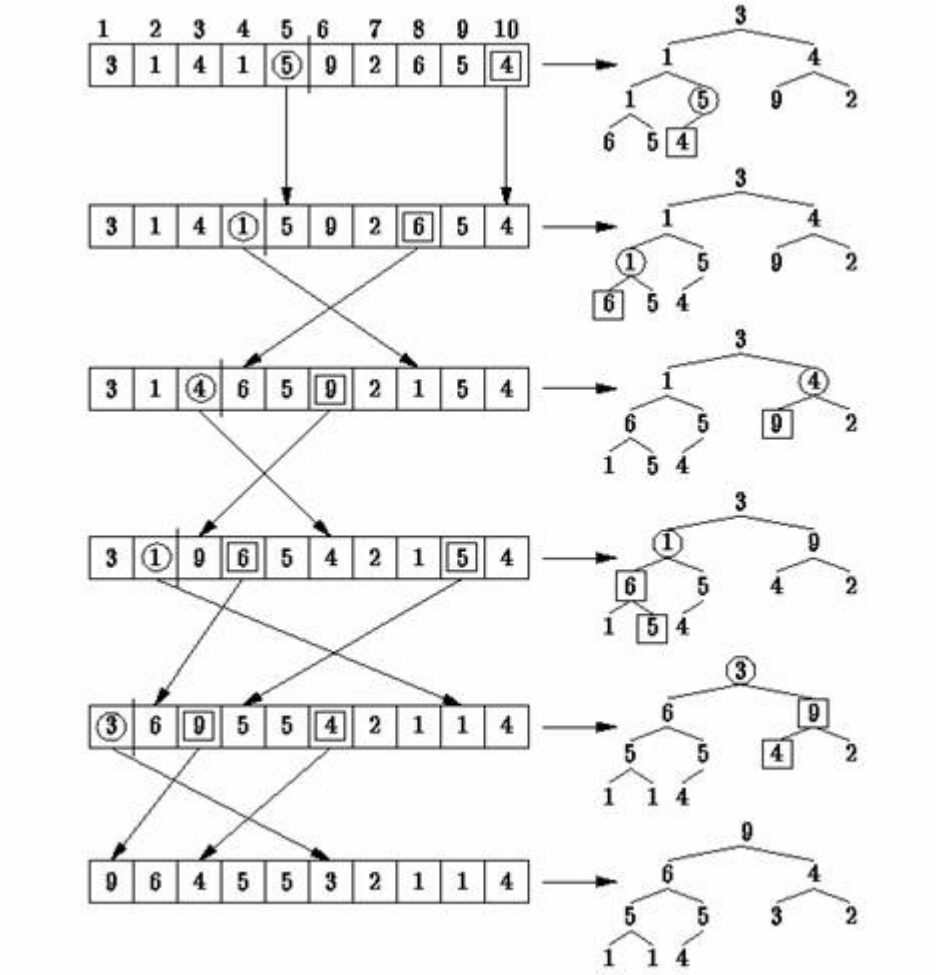
Heapsort

Táto metóda je založená na dátovej štruktúre zvanej halda, čo je vlastne binárny strom reprezentovaný poľom. N prvkov haldy je umiestnených na pozíciách 0 až N-1 poľa. Koreň haldy je na pozícii 0. Vo všeobecnosti potomkovia vrcholu na pozícii **i** poľa, sú umiestnení na pozíciách **2i** a **2i+1** , a rodič je umiestnený na pozícii **i/2** . V halde majú rodičovské vrcholy vždy väčšiu (resp. menšiu) hodnotu ako ich potomkovia. Algoritmus pozostáva z dvoch fáz:

• V prvej je neusporiadaná vstupná množina (pole) pretransformovaná na haldu. Pri vytváraní haldy platí nasledovne pravidlo: **Hodnota každého vrcholu, ktorý má rodiča, musí byť menšia alebo rovná ako hodnota rodiča.**

• V druhej vytvára usporiadanú postupnosť vyberaním maximálneho prvku z haldy. Pokračuje sa bodom 1. a tak ďalej až kým halda nie je prázdna





Umiestnime najvyšší vrchol na 0-tú pozíciu postupnosti. Ďalej, nech potomkovia hociktorého vrcholu sú umiestnení na miestach (2n) a (2n+1).

11. Usporadúvanie. metódy vonkajšieho usporadúvania: zlučovaním.

Vonkajšie usporadúvanie

Tyto metody se zabývají tříděním sekvenčních souborů. Připomeňme se nejdůležitější rozdíly ve srovnání s metodami pro třídění polí.

O souboru předpokládáme, že je uložen na vnějším, zpravidla magnetickém médiu se sekvenčním přístupem. To znamená, že musíme zpracovávat jeden záznam po druhém v pořadí, v jakém jsou v souboru uloženy.

Vedle toho zde předem neznáme rozsah tříděných dat, tj. počet *n* záznamů v souboru. Lze ale předpokládat, že je tak velký, že se soubor nevejde do operační paměti počítače.

Čtení záznamu ze souboru nebo uložení (zápis) do souboru budeme společně označovat jako *přístup do souboru*.Přístup do souboru trvá o několik řádů déle než porovnávání záznamů, takže je z hlediska efektivity algoritmů pro vnější třídění rozhodující. Proto si při rozboru algoritmů pro vnější třídění budeme všímat jen počtu přístupů do souboru.

Na druhé straně máme obvykle k dispozici dostatečné množství vnější paměti, takže s ní nemusíme příliš šetřit a budeme při třídění využívat pomocných souborů.

Skripta PT 2 -> str 215 – 220

metlak & pvi & maestro