Algoritmi și Structuri de Date

Lect. univ. dr. Horea Oros

horea.oros@gmail.com

Universitatea din Oradea

Facultatea de Științte

Agenda

- Introducere
- Colecții de obiecte: bag, queue, stack
- Analiza algoritmilor
- Algoritmi de sortare
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Merge Sort
  - Quick Sort

Agenda

- Ce este un algoritm?
- Algoritmi de sortare
- Algoritmi de căutare
- Structuri de date
  - Stiva
  - Coada
  - Arbori
  - Heap
- Algoritmi diverși

Introducere

- Ce este un algoritm?
  - Metodă pentru rezolvarea unei probleme, metodă ce poate fi implementată pe un calculator
  - Metoda este independentă de limbajul de programare folosit
  - Metoda este cea care stabilește pașii ce trebuie efectuați
  - Algoritmi + structuri de date
- Un algoritm se poate exprima:
  - în limbaj natural prin descrierea procedurii de rezolvare a problemei
  - printr-un program pentru calculator care implementează acea procedură.

Introducere

- Exemplu: Algoritmul lui Euclid (2300 ani)

```
public static int cmmdc(int a, int b)
        {
            if (b == 0)
                return a;
            else
                return cmmdc(b, a % b);
        }
```

Introducere

- Exprimăm programele în C# pentru că e mai ușor de verificat cerințele pe care le are algoritmul:
  - Finit
  - Determinist
  - Corect
- Folosim Visual Studio ca și mediu pentru dezvoltarea de aplicații
- GitHub pentru distribuirea programelor

Introducere

- Algoritmii necesită organizarea datelor asupra cărora operează
- Asta duce la *Structuri de date*
- Vom studia Algoritmi + Structuri de date
- Pentru a înțelege algoritmii trebuie să studiem și structuri de date
- Algoritmi simpli ⟹ structuri de date complicate
- Algoritmi complicați ⟹ structuri de date simple

Introducere

- Putem avea algoritmi naivi pentru rezolvarea unei probleme dar care sunt nefezabili computațional atunci când dimensiunea problemei este mare
- Ne vom concentra pe algoritmi eficienți pentru rezolvarea unor probleme
- Găsirea unui algoritm eficient este mult mai importantă decât o investiția în mai multă putere de calcul
- Mulți algoritmi sunt deja implementați în diverse biblioteci. De ex. **sort**
- Crearea propriilor implementări ne va ajuta să înțelegem mai bine funcționarea acestora și să luăm decizii corecte în alegerea algoritmului/structurii de date folosite într-un anumit context.

Introducere

- Căutare binară

```
public static int rank(int key, int[] a)
        {
            int lo, hi, mid;
            lo = 0;
            hi = a.Length - 1;
            while (lo <= hi)
            {
                mid = lo + (hi - lo) / 2;
                if (key < a[mid])  hi = mid - 1;
                else if (key > a[mid])   lo = mid + 1;
                else   return mid;
            }
            return -1;
        }
```

Introducere

- **Bibliografie:**
- *Algorithms, 4th Edition* by Robert Sedgewick and Kevin Wayne
  - http://algs4.cs.princeton.edu/home/
  - https://www.coursera.org/course/algs4partI
- *Introduction to Algorithms, 3rd Edition, 2009* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- www.infoarena.ro
- http://uva.onlinejudge.org/
- https://github.com/HoreaOros/ASD2015 (implementări C# ale algoritmilor/structurilor de date din curs

Modelul de programare

- Componentele de bază ale oricărui limbaj de programare modern
  - Tipurile de date primitive (bool, int, char, float, double etc.)
  - Instrucțiunile (ne permit să definim calcule): declarare, atribuire, condiționale, repetitive, apel, return
  - Tablouri – agregarea mai multor valori de același tip
  - Metode statice – încapsularea și reutilizarea codului pentru a dezvolta programele ca seturi de module independente
  - String - șiruri de caractere
  - Intrare/ieșire – comunicarea dintre program și restul lumii
  - Abstractizarea datelor – POO – crearea de noi tipuri de date non-primitive (clase)

Colecții de obiecte: bag, queue, stack

- Bag, Queue, Stack: structuri de date fundamentale
- Modul în care sunt reprezentate datele într-o colecție influențează direct eficiența diferitelor operații
- Generics și iterații
- Structuri de date înlănțuite – permit implementări ce realizează obiectivele de eficiență
- Înțelegerea listelor înlănțuite este un prim pas în studiul algoritmilor și structurilor de date
- Vom investiga mai multe modalități de reprezentare a valorilor și de implementare a operațiilor specifice

Colecții de obiecte: Bag

- API pentru Bag:

```
class Bag<Item>
    public Bag() // Crearea unui bag gol
    public void add(Item item) // Adaugarea unui element
    public bool isEmpty() // Este gol?
    public int size() // Numarul de elemente din Bag
```

Colecții de obiecte: Queue

- API pentru Queue:

```
class Queue<Item>
    // Crearea unei cozi fara nici un element
    public Queue()
     // Adaugarea unui element in coada
    public void enqueue(Item item)
    // Eliminarea elementului care a fost adaugat cel mai demult
    public Item dequeue()
    // Este goala coada?
    public bool isEmpty()
     // Numarul de elemente din coada
    public int size()
```

Colecții de obiecte: Stack

- API pentru Stack:

```
/// Stiva LIFO
class Stack<Item>
    // Crearea unei stive fara nici un element
    public Stack()
    /// Adaugarea unui element in stiva
    public void push(Item item)
    /// Eliminarea ultimului elementului adaugat
    public Item pop()
    /// Este goala stiva?
    public bool isEmpty()
    /// Numarul de elemente din stiva
    public int size()
```

Colecții de obiecte - generics

- O colecție trebuie să o putem folosi pentru orice tip de date
- Mecanismul care ne permite asta se numește: generics sau tipuri parametrizate
- Notația `<Item>` după numele clasei definește Item ca fiind parametru de tip, un înlocuitor pentru un tip concret de date
- La implementarea tipului colecție nu știm care este tipul concret de date
- Clientul va putea folosi clasa colecție pentru orice tip de date
- Codul client este cel care furnizează tipul de date concret la crearea obiectului colecție

Colecții de obiecte - generics

- Putem scrie cod client de forma:

```
Stack<String> stack = new Stack<String>();
stack.push("Test");
...
String next = stack.pop();
```

- sau

```
Queue<Date> queue = new Queue<Date>();
queue.enqueue(new Date(31, 12, 1999));
...
Date next = queue.dequeue();
```

Colecții de obiecte - generics

- Încercarea de a adăuga un obiect de alt tip (necompatibil) la colecție duce la eroarea la compilare
- Generics – ne permite cod type-safe; erorile pot fi detectate la compilare
- Fără generics trebuie să definim o clasă colecție diferită pentru fiecare tip de date ale cărui valori am dori să le păstrăm într-o colecție

Colecții de obiecte – IEnumerable<Items>

- TODOs
- http://msdn.microsoft.com/en-us/library/vstudio/ee5kxzk0(v=vs.100).aspx

Colecții de obiecte – Bag

- Bag colecție din care nu se pot elimina elementele
- Permite clientului doar să adune elemente și să le acceseze
- Ordinea elementelor din Bag nu e importantă
- Iterarea peste elementele din Bag nu trebuie să se facă în ordinea în care au fost adăugate
- BagClient.cs

Colecții de obiecte – Queue - FIFO

- Se bazează pe principiul First In – First Out (primul venit – primul servit)
- Un principiu întâlnit în viața de zi cu zi: permite un serviciu echitabil
- Cel care a așteptat cel mai mult este cel care este servit primul
- Au un rol central în multe aplicații
- Folosim coada pentru a păstra ordinea relativă a elementelor
- QueueClient.cs

Colecții de obiecte – Stack- LIFO

- Stiva se bazează pe principiul Last In – First Out (Ultimul venit – primul servit)
- Structură de date fundamentală în Informatică
- De ex. Hyperlink-urile pe care le urmăm în browser se pun într-o stivă pentru putea reveni la ele cu *Back*.
- StackClient.cs

Colecții de obiecte – Stack- LIFO Aplicație

- Să se evalueze o expresie aritmetică de forma:
  - (1 + ((2 + 3) * (4 * 5)))
- Expresie în care sunt puse toate parantezele (nu ținem cont de prioritatea operatorilor.
- *ExpAritm* := *Operand* | (*ExpAritm* **operator** *ExpAritm*)
- Indicii de rezolvare:
  - Algoritm datorat lui E.W. Dijkstra (1960)
  - Folosim două stive: una pentru operanzi și una pentru operatori

Colecții de obiecte – Stack- LIFO Aplicație

- *ExpAritm* := *Operand* | (*ExpAritm* **operator** *ExpAritm*)
- Indicii de rezolvare:
  - Operand – se pune pe stiva de operanzi
  - Operator – se pune pe stiva de operatori
  - (- se ingoră
  -) – se scoate un operator, se scot cei doi operanzi, se aplică operatorul asupra celor doi operanzi iar rezultatul se pune înapoi în stiva de operanzi
  - După procesarea ultimei) în stiva de operanzi rămâne o singură valoare = valoarea expresiei
  - EvaluareExpresie.cs

Colecții de obiecte
Implementare

- Stivă de dimensiune fixă

```
private Item[] data;
private int count = 0;
private int capacity = 100;
...
public Stack(int capacity)
{
    this.capacity = capacity;
    data = new Item[capacity];
}
```

Colecții de obiecte
Implementare - stiva

```
public void push(Item item)
        {
            if (count < capacity - 1)
            {
                data[count++] = item;
            }
            else
                throw new StackFullException();

        }
```

Colecții de obiecte
Implementare - stiva

```
public Item pop()
        {
            if (!isEmpty())
            {
                return data[--count];
            }
            else
                throw new StackEmptyException();
        }
```

Colecții de obiecte
Implementare - stiva

```
public Item peek()
        {
            if (!isEmpty())
            {
                return data[count - 1];
            }
            else
                throw new StackEmptyException();
        }
```

Colecții de obiecte
Implementare - stiva

```
public bool isEmpty()
        {
            return count == 0;
        }
public int size()
        {
            return count;
        }
```

Colecții de obiecte – avantaje/dezavantaje

- Orice implementare pentru structurile de date poate avea avantaje și dezavantaje
- Generics oferă avantaje evidente pe care le-am discutat deja
- Dacă datele sunt într-un tablou de dimensiune fixă avem o limitare inerentă – când tabloul este plin nu mai putem introduce elemente noi (... și lansăm excepție) – numărul maxim de elemente trebuie estimat de la început (imposibil)
- O soluție este crearea unor structuri de date în care tabloul se poate redimensiona
- Avantaj la utilizarea tablourilor: fiecare operație asupra colecției se face in timp constant
- Dezavantaj: redimensionarea colecției este o operație ineficientă
- ResizingStack.cs

Colecții de obiecte
Implementare – stiva redimensionabilă

```
private void resize(int max)
        {
            Item[] temp = new Item[max];
            for(int i = 0; i < count; i++)
            {
                temp[i] = data[i];
            }
            data = temp;
            capacity = max;
        }
```

Colecții de obiecte
Implementare – stiva redimensionabilă

- Dacă stiva este plină îi dublăm capacitatea

```
public void push(Item item)
        {
            if (count == capacity)
                resize(2 * capacity);
            data[count++] = item;
        }
```

Colecții de obiecte
Implementare – stiva redimensionabilă

- Dacă stiva e destul de goală atunci se poate micșora (înjumătăți capacitatea)
- Ce ar însemna destul de goală?

```
public Item pop()
        {
            Item item;
            if (!isEmpty())
            {
                item =  data[--count];
                data[count] = default(Item);
                if (count == capacity / 4) resize(capacity / 2);
                return item;
            }
            else  throw new StackEmptyException();
        }
```

Liste înlănțuite
StackLL.cs

- Definiție: O listă înlănțuită este o structură de date recursivă care este ori goală (null) ori este o referință la un nod care conține un item generic și o referință la o altă listă înlănțuită (mai scurtă)
- Nodul este o entitate abstractă care poate conține orice tip de date
- Vom crea o clasă încuibată (nested) in clasa *Stack*:

```
private class Node
        {
            public Item item;
            public Node next;
        }
```

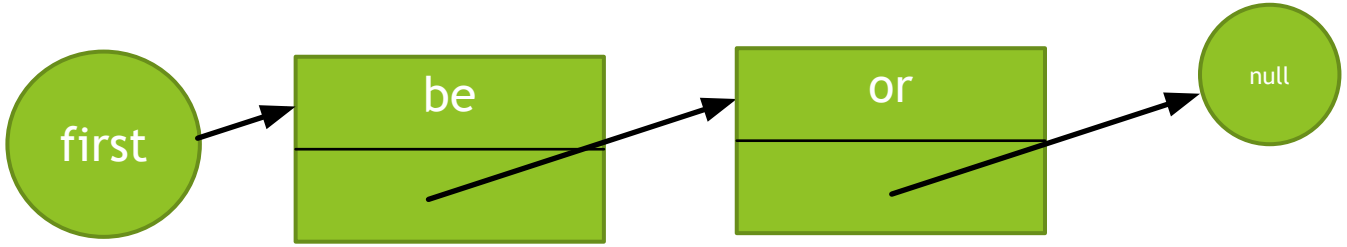
Liste înlănțuite
Implementarea unei stive

- Vom crea un obiect de tip Node astfel:

```
Node first = new Node();
```

- Pentru accesa componentele vom folosi sintaxa:

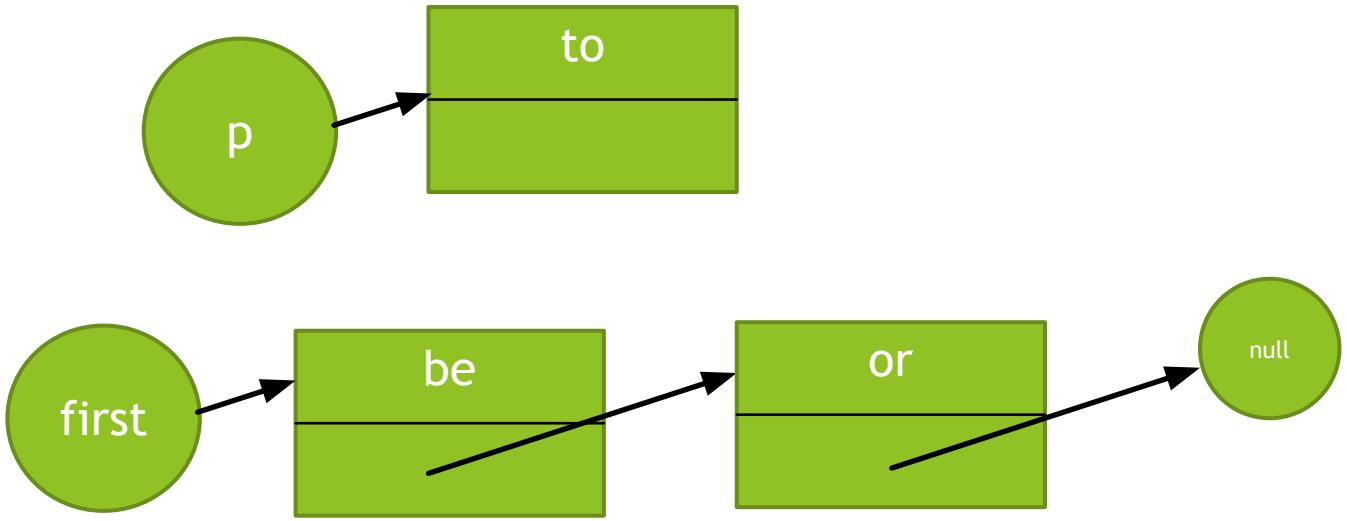
```
first.item
first.next
```



Liste înlănțuite
Implementarea unei stive

- Adăugarea unui element la începutul listei

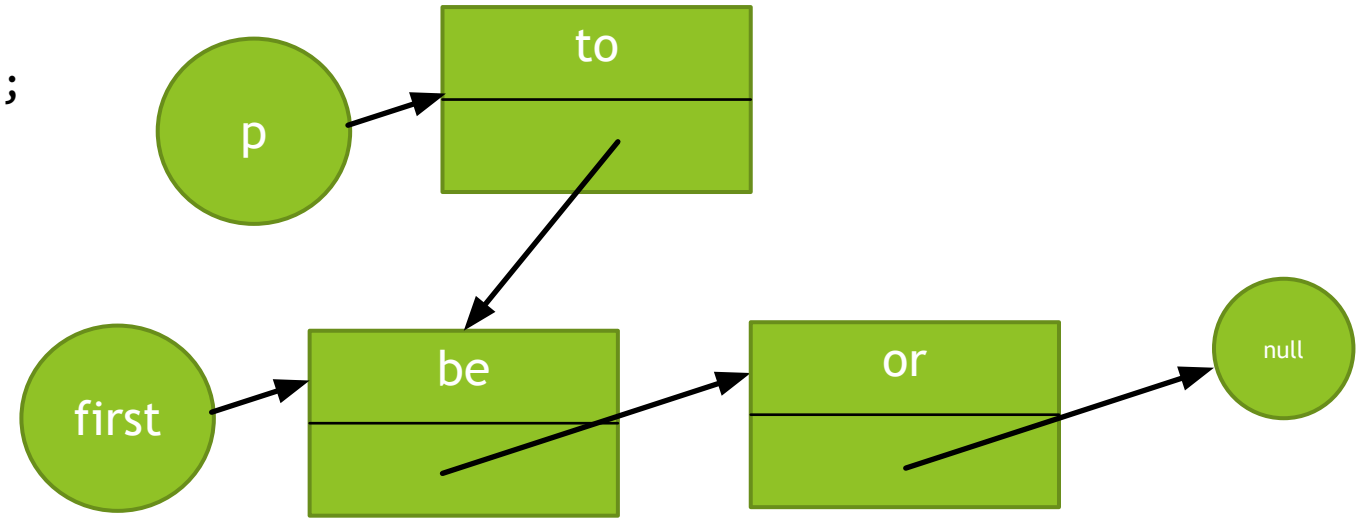
```
Node p = new Node();
p.item = item;
```



Liste înlănțuite
Implementarea unei stive

- Adăugarea unui element la începutul listei

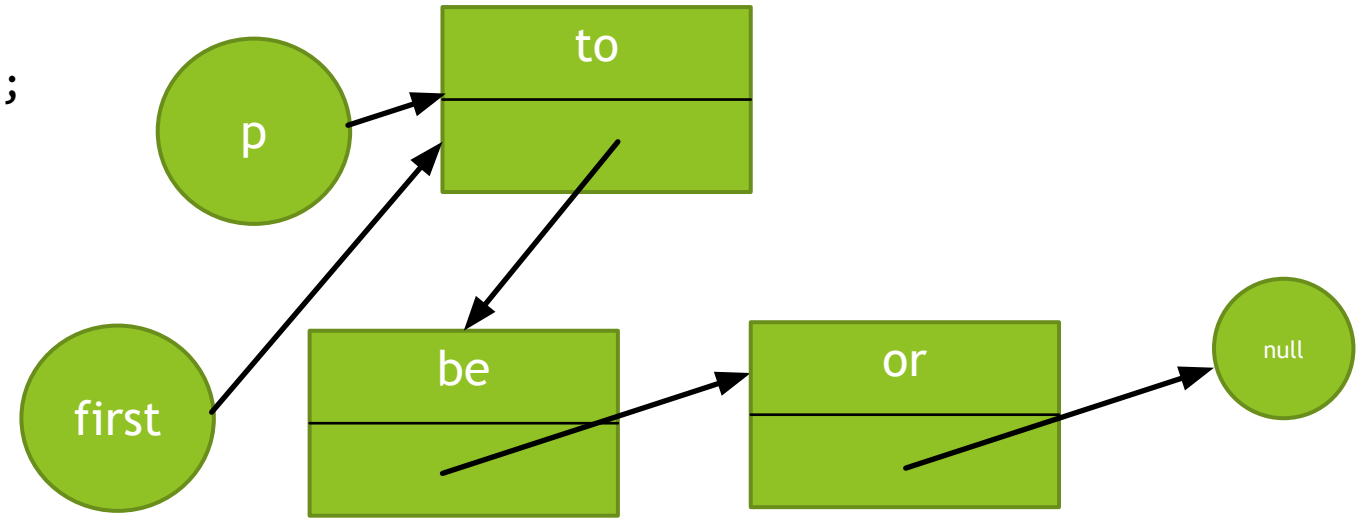
```
Node p = new Node();
p.item = item;
p.next = first;
```



Liste înlănțuite
Implementarea unei stive

- Adăugarea unui element la începutul listei

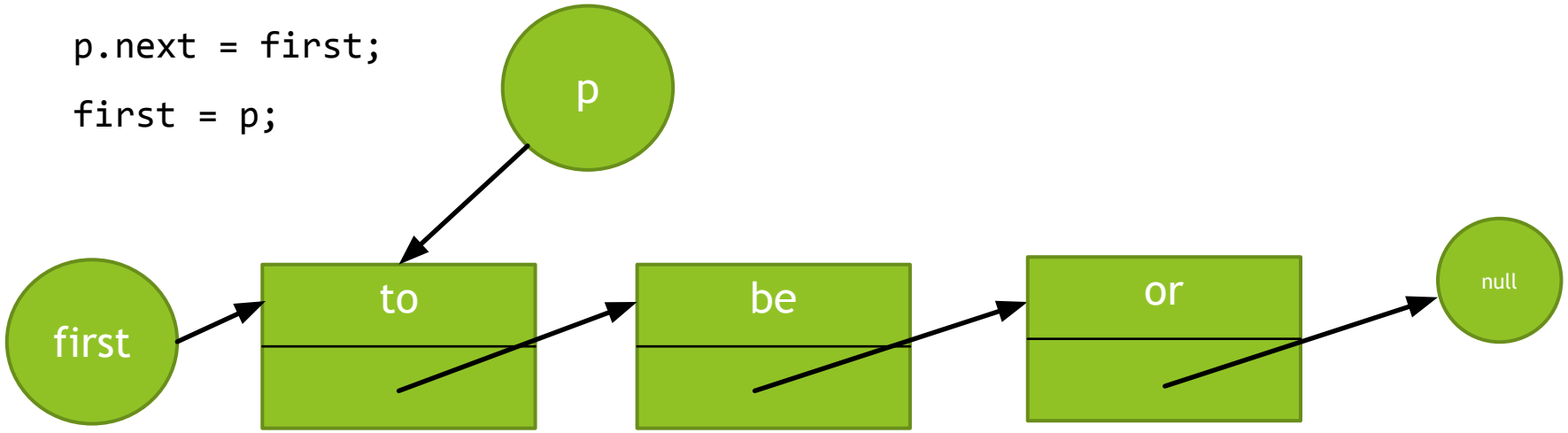
```
Node p = new Node();
p.item = item;
p.next = first;
first = p;
```



Liste înlănțuite
Implementarea unei stive

- Adăugarea unui element la începutul listei

```
Node p = new Node();
p.item = item;
p.next = first;
first = p;
```



Liste înlănțuite
Implementarea unei stive

- Ștergerea elementului de la începutul listei

```
public Item pop()
        {
            if(!isEmpty())
            {
                Node temp = first;
                first = first.next;
                return temp.item;
            }
            else
                throw new StackEmptyException();
```

Liste înlănțuite

- Alte operații:
  - Adăugarea unui nod la sfârșitul listei
  - Inserarea/Eliminarea unui nod oarecare din interiorul listei
  - Traversarea listei
- Parcurgerea listei este o operație ineficientă – timpul necesar este proporțional cu dimensiunea listei
- O soluție standard pentru realizarea operațiilor de inserare/ștergere de noduri aleatoare este crearea unei liste dublu-înlănțuite (temă)

Liste înlănțuite

- Traversarea listei

```
public IEnumerator<Item> GetEnumerator()
        {
            Node p = first;
            while (p != null)
            {
                yield return p.item;
                p = p.next;
            }
        }
```

Liste înlănțuite

- Implementarea realizată respectă următoarele cerințe:
  - Poate fi folosită cu orice tip de date (generics)
  - Spațiul de memorie ocupat este întotdeauna proporțional cu numărul de elemente din listă
  - Timpul necesar pentru realizarea operațiilor (push/pop) este independent de dimensiunea colecției
- Pentru a implementa o coadă cu ajutorul unei liste înlănțuite:
  - Păstrăm două referințe: la începutul și la sfârșitul cozii
- Listele înlănțuite reprezintă o alternativă fundamentală la vectori pentru structurarea colecțiilor de date:
  - LISP (1950) – listele sunt structurile fundamentale pentru programe și date

Colecții de obiecte

- Aceste implementări de clase colecție oferă un nivel de abstractizare ce ne permite să scriem un client compact
- Înțelegerea acestor tipuri de date abstracte este esențială, fiind o introducere în studiul algoritmilor și a structurilor de date
  - Folosim aceste tipuri de date ca elemente ale altor tipuri de date mai complexe
  - Ilustrează legătura dintre algoritmi și structuri de date
  - Ne vom concentra pe tipuri de date care oferă operații mai puternice asupra colecțiilor de obiecte – iar aceste implementări sunt puncte de plecare

Colecții de obiecte

Structura de date	Avantaj	Dezavantaj
Tablou/vector	Indexul oferă acces rapid la element	Dimensiunea trebuie cunoscută la inițializare
Listă înlănțuită	Spațiul folosit este proporțional cu dimensiunea colecției	Pentru a accesa un element avem nevoie de o referință la element

Colecții de obiecte

- Aceste colecții pot fi extinse în mai multe moduri
  - Liste înlănțuite cu mai multe legături - arbori
- Colecțiile pot fi compuse
  - Tablouri de stive
  - Cozi de stive
  - Stive de tablouri
  - Etc.

Colecții de obiecte
Exerciții

- Se efectuează o serie de operații push()/pop() intercalate. pop() afișează valoarea extrasă. Valorile care se inserează sunt in ordine 0…9. Care din următoarele secvențe nu poate fi afișată?

a. 4 3 2 1 0 9 8 7 6 5 ✅

b. 4 6 8 7 5 3 2 9 0 1 ⛔

c. 2 5 6 7 4 8 9 3 1 0 ✅

d. 4 3 2 1 0 5 6 7 8 9 ✅

e. 1 2 3 4 5 6 9 8 7 0 ✅

f. 0 4 6 5 3 8 1 7 2 9 ⛔

g. 1 4 7 9 8 6 5 3 0 2 ⛔

h. 2 1 4 3 6 5 8 7 9 0 ✅

Colecții de obiecte
Exerciții

- Se dă un șir de caractere de tipul `[()]{}{[()()]()}`. Se cere să se determine dacă parantezele sunt închise corect. De ex. pentru `[(])` nu sunt închise corect.
- Ce face secvența de mai jos:

```
Stack<int> stack = new Stack<int>();
while (N > 0)
    {
        stack.push(N % 2);
        N = N / 2;
    }
foreach (int d in stack)
    Console.Write(d);
```

Colecții de obiecte
Exerciții

- Se efectuează o serie de operații enqueue()/dequeue() intercalate asupra unei cozi. Valorile care se inserează sunt in ordine 0…9. Care din următoarele secvențe nu poate fi afișată?

a. 0 1 2 3 4 5 6 7 8 9 

b. 4 6 8 7 5 3 2 9 0 1 

c. 2 5 6 7 4 8 9 3 1 0 

d. 4 3 2 1 0 5 6 7 8 9 

- Mai multe exerciții – Sedgwick, *Algorithms*, pg. 161

Analiza algoritmilor

- Folosim algoritmi/calculatoare pentru a rezolva probleme dificile
- Două întrebări tipice:
  - Cât timp va rula programul?
  - Câtă memorie va folosi programul?
- Răspunsul depinde de mai mulți factori:
  - Proprietățile sistemului de calcul folosit
  - Datele procesate
  - Programul care rezolvă problema (implementarea unui algoritm)

Analiza algoritmilor

- Vom obține răspunsuri aplicând metoda științifică
  - Vom aplica tehnici matematice pentru a obține modele concrete pentru costul algoritmului
  - Vom face studii experimentale pentru a valida modelele create
- Metoda științifică
  - *Observare* – a unei trăsături de regulă prin măsurători exacte
  - *Ipoteză* – a unui model pe baza observațiilor făcute
  - *Prezicere* - evenimente pe baza ipotezei
  - *Verificare* – a prezicerilor prin alte observații
  - *Validare* – repetăm pașii de mai sus până când corespund observațiile cu ipotezele

Analiza algoritmilor

- Experimentele trebuie să fie repetabile
- Ipotezele trebuie să le putem falsifica – pentru a ști când o ipoteză este greșită
- Nu putem ști niciodată că o ipoteză este corectă; putem doar valida faptul că ea corespunde cu observațiile făcute
- Observația
  - Se face foarte simplu – prin rularea programului și înregistrarea timpului de rulare

Analiza algoritmilor

- Orice problemă are o *dimensiune*
  - Dimensiunea tuturor valorilor pe care le procesează
  - O valoarea numerică dată la intrare – *N*.
- Orice problemă va avea și un *timp de execuție*
  - Crește o dată cu dimensiunea problemei
- Exemplu: Se dau n numere. Se cere să se determine numărul de triplete a căror sumă e 0. ThreeSum.cs

Analiza algoritmilor

- Pentru n – 1000, 2000, 4000, 8000 – cât timp durează rularea metodei count()?

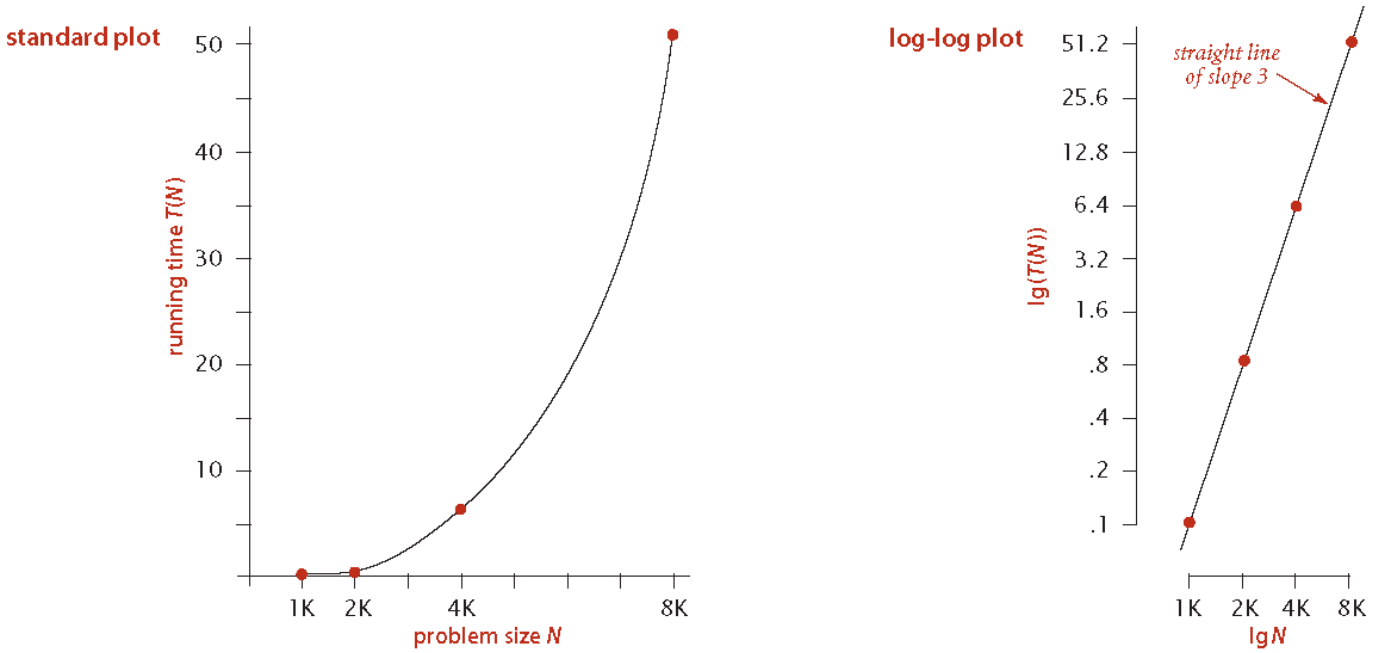
```
public static int count(int[] arr)
        {
            int contor = 0, n;
            n = arr.Length;
            for(int i = 0; i < n; i++)
                for(int j = i + 1; j < n; j++)
                    for(int k = j + 1; k < n; k++)
                        if (arr[i] + arr[j] + arr[k] == 0)
                        {
                            contor++;
                        }
            return contor;
        }
```

Analiza algoritmilor

N	Triplete cu suma zero	Timpul de executie
1000	70	00:00:00.1232575
2000	528	00:00:00.8773364
4000	4039	00:00:06.8908254
8000	32074	00:00:54.5196087

Analiza algoritmilor

- Pentru a măsura timpul de execuție putem folosi clasa `StopWatch` din namespace-ul `System.Diagnostic`
- http://en.wikipedia.org/wiki/Log-log_plot



Analysis of experimental data (the running time of `ThreeSum.count()`)

Analiza algoritmilor

- Pe graficul log-log se observ că datele sunt pe o dreaptă de pantă 3
- Ecuația unei astfel de drepte este:
  - log(T(N)) = 3log N + log a, unde a este o constantă
  - Echivalentă cu T(N) = a N^3
  - T(8000) = 54.51 = a * (8000^3)
  - Deci a = 10,64 * (10 ^ (-9))
  - Obținem ecuația T(N) = 10,64 * (10 ^ (-9)) * (N^3) pentru estimarea timpului de execuție
- Cu o astfel de formulă putem prezice cât timp va rula programul pentru N = 16000 și să verificăm acest lucru.
- O dreaptă pe graficul log-log duce la concluzia că datele corespund ecuației:
  - T(N) = a N^b

Analiza algoritmilor
modele matematice

- D.Knuth a postulat faptul că timpul de rulare al unui program este determinat de doi factori:
  - Costul execuției fiecărei instrucțiuni (proprietate a sistemului de calcul, compilatorului și sistemului de operare)
  - Frecvența de execuție a fiecărei instrucțiuni (proprietate a programului și a datelor de intrare)
- Dacă știm câte instrucțiuni se execută și cât durează fiecare instrucțiune putem înmulți/însuma aceste valori și obținem timpul total de rulare
- Numărul de execuții al unor instrucțiuni e simplu de determinat
- Pentru altele e mai complicat. De ex. de câte ori se execută instrucțiunea if în ThreeSum?
  - Combinari de N luate câte 3 = N(N - 1)(N - 2) / 6

Analiza algoritmilor
modele matematice

- N(N - 1)(N - 2) / 6 = N^3 / 6 – N^2 / 2 + N / 3
- Termeni de ordin mic sunt nesemnificativi când N are valoare mare prin urmare pot fi ignorați
- Putem folosi notația ~ (tilda) pentru a simplifica formulele matematice de acest tip
- Definiție:
  - Vom folosi notația ~f(N) pentru a reprezenta o funcție care atunci când e împărțită la f(N) rezultatul tinde la 1 atunci când N crește
  - Folosim notația g(N) ~ f(N) pentru a indica faptul că g(N) / f(N) tinde la 1 când N crește.

Analiza algoritmilor modele matematice

- Aproximări tilda

Funcția	Aproximare tilda	Ordinul de creștere
$N^3 / 6 - N^2 / 2 + N/3$	$\sim N^3 / 6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

Analiza algoritmilor modele matematice

- Ordinul de creștere

Descriere	Funcție
Constant	1
Logaritmic	Log N
Liniar	N
Linearitmic	N Log N
Quadratic/pătratic	N^2
Cubic	N^3
Exponențial	2^N

Analiza algoritmilor modele matematice

- Aproximările tilda vor fi de forma:
  - g(N) ~ a f(N)
  - unde f(N) = N^b (log N)^c
  - a, b, c sunt constante
- Baza logaritmului nu e importantă întrucât e absorbită de constante

Analiza algoritmilor
modele matematice

- Proprietatea A: Ordinul de creștere pentru timpul de execuție al ThreeSum este N^3
- Justificare:
  - fie T(N) timpul de execuție.
  - Modelul descris sugerează cu T(N) ~ aN^3. Unde a este o constantă de depinde de sistemul de calcul pe care rulează programul.
  - Experimente pe diverse sisteme de calcul validează această aproximare
- Proprietate = ipoteză care trebuie validată prin experimente
- Rezultatul analizei matematice este același cu rezultatul analizei experimentale. Prin urmare atât experimentele cât și modelul matematic sunt validate

Analiza algoritmilor modele matematice

- Funcții uzuale în analiza algoritmilor

description	notation	definition
floor	$\lfloor x \rfloor$	largest integer not greater than x
ceiling	$\lceil x \rceil$	smallest integer not smaller than x
natural logarithm	$\ln N$	$\log_e N$ (x such that $e^x = N$)
binary logarithm	$\lg N$	$\log_2 N$ (x such that $2^x = N$)
integer binary logarithm	$\lfloor \lg N \rfloor$	largest integer not greater than $\lg N$ (# bits in binary representation of N) – 1
harmonic numbers	H_N	$1 + 1/2 + 1/3 + 1/4 + \ldots + 1/N$
factorial	$N!$	$1 \times 2 \times 3 \times 4 \times \ldots \times N$

Analiza algoritmilor modele matematice

- Aproximări uzuale

description	approximation
harmonic sum	$H_N = 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/N \sim \ln N$
triangular sum	$1 + 2 + 3 + 4 + \ldots + N \sim N^2/2$
geometric sum	$1 + 2 + 4 + 8 + \ldots + N = 2N - 1 \sim 2N$ when $N = 2^n$
Stirling's approximation	$\lg N! = \lg 1 + \lg 2 + \lg 3 + \lg 4 + \ldots + \lg N \sim N \lg N$
binomial coefficients	$\binom{N}{k} \sim N^k/k!$ when k is a small constant
exponential	$(1 - 1/x)^x \sim 1/e$

Analiza algoritmilor
model de cost

- Modelul de cost = definește operațiile elementare folosite de un algoritm
- Modelul de cost pentru ThreeSum = numărul de accese la elementele vectorului
- Propoziția B: algoritmul ThreeSum folosește ~N^3 / 2 accese la elementele vectorului pentru a calcula numărul tripletelor a căror sumă e zero.
  - Demonstrație: algoritmul accesează fiecare din cele 3 elemente ale celor ~N^3 / 6 triplete
- Propoziție = adevăr matematic despre algoritmi relativ la modelul de cost

Analiza algoritmilor
model de cost

- Crearea unui model matematic pentru timpul de execuție:
  - Crearea unui model pentru datele de intrare – definirea dimensiunii problemei
  - Identificarea buclei interioare
  - Definirea unui model de cost ce include operații în bucla interioară
  - Determinarea frecvența de execuție a acelor operații pentru datele de intrare
- BinarySearch
  - Modelul pentru datele de intrare: vectorul de dimensiune N
  - Bucla interioară: instrucțiunea din singura instrucțiune `while`
  - Modelul de cost: operația relațională (de comparație)
  - Frecvența de execuție: cel mult `log N + 1`

Analiza algoritmilor
Clasificarea algoritmilor după ordinul de creștere

- Constant
  - Execută un număr fix de operații. Timpul de execuție nu depinde de N (dimensiunea problemei)
  - Ordinul de creștere = 1
  - Ex. a = b + c;
  - O instrucțiune simplă
  - Descriere: instrucțiune
- Logaritmic
  - Exemplul clasic este căutarea binară
  - Ordinul de creștere = log N
  - Baza logaritmului nu e relevantă
  - Descriere: Înjumătățire

Analiza algoritmilor
Clasificarea algoritmilor după ordinul de creștere

- Liniar
  - Procesează fiecare element din intrare în timp constant
  - Bazate pe o singură instrucțiune for
  - Ordinul de creștere = N
  - Ex. determinarea maximului dintr-un vector
  - Descriere: buclă
- Linearitmic
  - Ordinul de creștere = N log N
  - Exemple tipice: Merge Sort, Quick Sort
  - Descriere: divide et impera

Analiza algoritmilor
Clasificarea algoritmilor după ordinul de creștere

- Pătratic
  - Bazate pe două instrucțiuni for incluse una într-alta
  - Ordinul de creștere = N^2
  - Ex. algoritmii de sortare elementari: Insertion Sort, Selection Sort, Bubble Sort
  - Descriere: buclă dublă
- Cubic
  - Ordinul de creștere = N^3
  - Exemple tipice: verificarea tuturor tripletelor
  - Descriere: buclă triplă

Analiza algoritmilor
Clasificarea algoritmilor după ordinul de creștere

- Exponențial
  - Ordinul de creștere = a^N (unde a este o constantă mai mare decât 1)
  - Foarte lent
  - Există o clasă mare de probleme pentru care cel mai bun algoritm cunoscut este exponențial
  - În practică poate rezolva doar probleme dimensiune mică
- Sunt și alte exprimări ale ordinului de creștere: N^2 log N sau N^(3/2)
- Pentru a determina astfel de ordine de creștere e nevoie de o serie de unelte matematice
- Algoritmii cubici sau pătratici nu sunt fezabili pentru rezolvarea unor probleme de dimensiune mare
- Ne vom concentra pe algoritmi fundamentali liniari, logaritmici sau linearitmici care pot rezolva problema mult mai repede.

Analiza algoritmilor
Implementare mai rapidă pentru ThreeSum

- Încălzire cu TwoSum – TwoSum.cs
- Implementare ineficientă – algoritm pătratic – N^2

```
public static int count(int[] arr)
        {
            int contor = 0, n;
            n = arr.Length;
            for (int i = 0; i < n; i++)
                for (int j = i + 1; j < n; j++)
                    if (arr[i] + arr[j] == 0)
                     contor++;
                 return contor;
        }
```

Analiza algoritmilor
Implementare mai rapidă pentru ThreeSum

- Încălzire cu TwoSum – TwoSum.cs
- Implementare eficientă – soluție linearitmică- N Log N

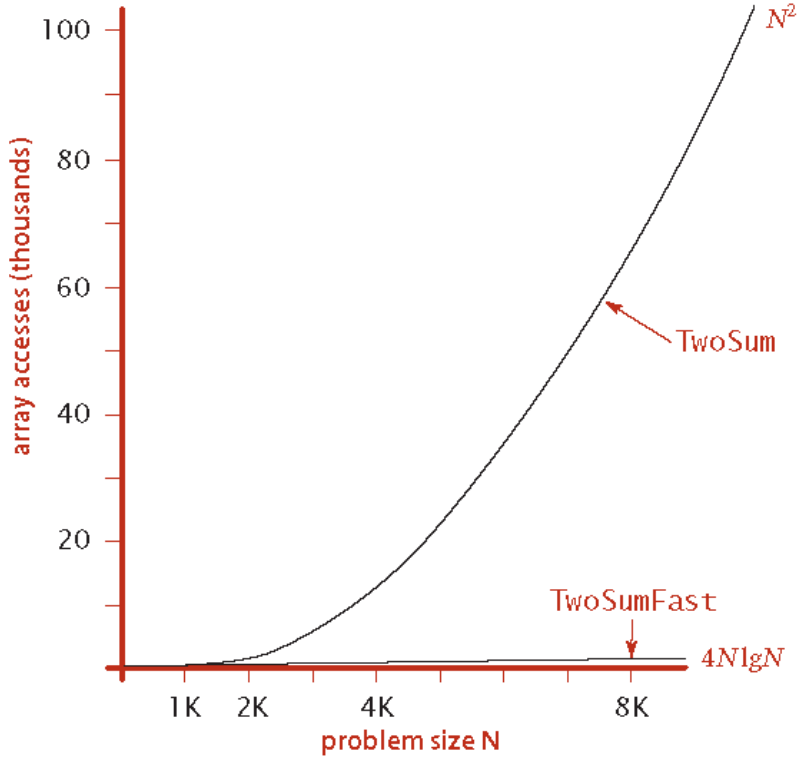
```
public static int countFast(int[] arr)
        {
            int contor = 0, n;
            n = arr.Length;
            Array.Sort(arr);
            for (int i = 0; i < n; i++)
                if (BinarySearch.rank(-arr[i], arr) > i)
                    contor++;
            return contor;
        }
```

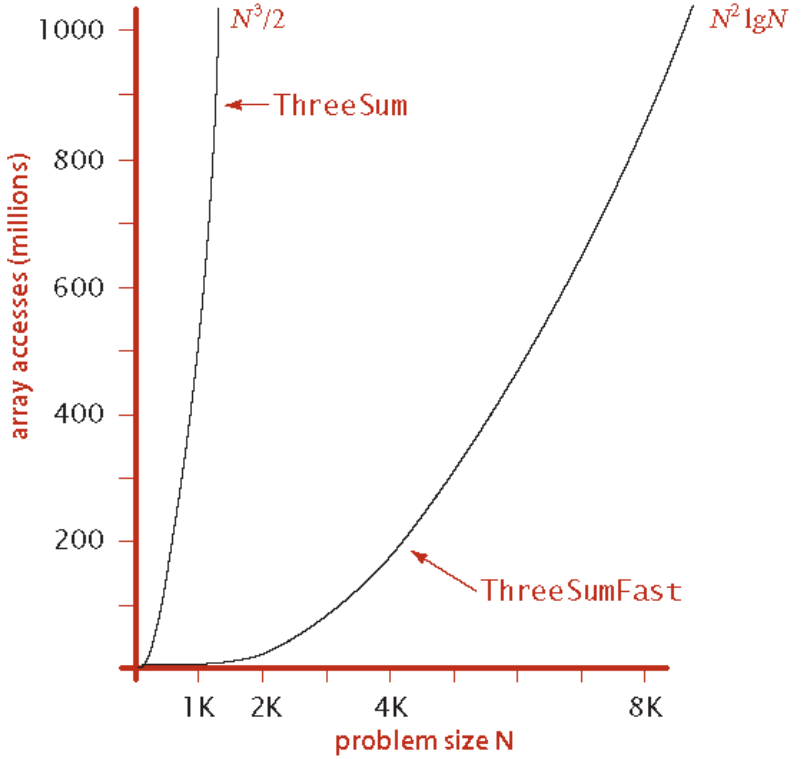
Analiza algoritmilor
Implementare mai rapidă pentru ThreeSum

- ThreeSum eficient - N^2 log N

```
public static int countFast(int []arr)
        {
            int contor = 0,  n;
            n = arr.Length;
            Array.Sort(arr);
            for (int i = 0; i < n; i++)
                for (int j = i + 1; j < n; j++)
                    if (BinarySearch.rank(-arr[i] - arr[j], arr) > j)
                        contor++;
            return contor;
        }
```

Analiza algoritmilor
Implementare mai rapidă pentru ThreeSum





Analiza algoritmilor
Experimente de dublare

- Vom genera valori asupra cărora să opereze algoritmul
- Vom rula programul `DoublingRatio` pentru a calcula raportul dintre timpul de rulare curent și cel anterior
- Vom rula experimente până când raportul se apropie de o limită 2^b
- La fiecare experiment numărul elementelor din intrare se dublează față de experimentul anterior
- Nu funcționează pentru orice algoritm dar merge pentru foarte multe
- Ordinul de creștere pentru timpul de execuție se apropie de N^b
- Pentru a anticipa timpul de execuție înmulțim timpul de execuție anterior cu 2^b și dublăm N

Analiza algoritmilor

- Propoziție: dacă T(N) ~ a N^b log N atunci T(2N) / T(N) ~ 2^b
- Demonstrație:
  - T(2/N) / T(N) = a(2N)^b log 2N / a N^b log N
  - = 2^b (1 + log 2 / log N)
  - ~ 2^b
- Acest rezultat e valabil pentru toate ordinele de creștere *cu excepția* celui exponențial

Analiza algoritmilor

- Notațiile O, o, Theta, theta, Omega, omega
- TODO

Analiza algoritmilor
Exerciții

- Care este ordinul de creștere pentru secvența de mai jos?

```
int sum = 0;
for (int n = N; n > 0; n /= 2)
    for (int i = 0; i < n; i++)
        sum++;
```

Analiza algoritmilor
Exerciții

- Care este ordinul de creștere pentru secvența de mai jos?

```
int sum = 0;
for (int i = 1 i < N; i *= 2)
    for (int j = 0; j < i; j++)
        sum++;
```

Analiza algoritmilor
Exerciții

- Care este ordinul de creștere pentru secvența de mai jos?

```
int sum = 0;
for (int i = 1 i < N; i *= 2)
    for (int j = 0; j < N; j++)
        sum++;
```

Analiza algoritmilor
Probleme

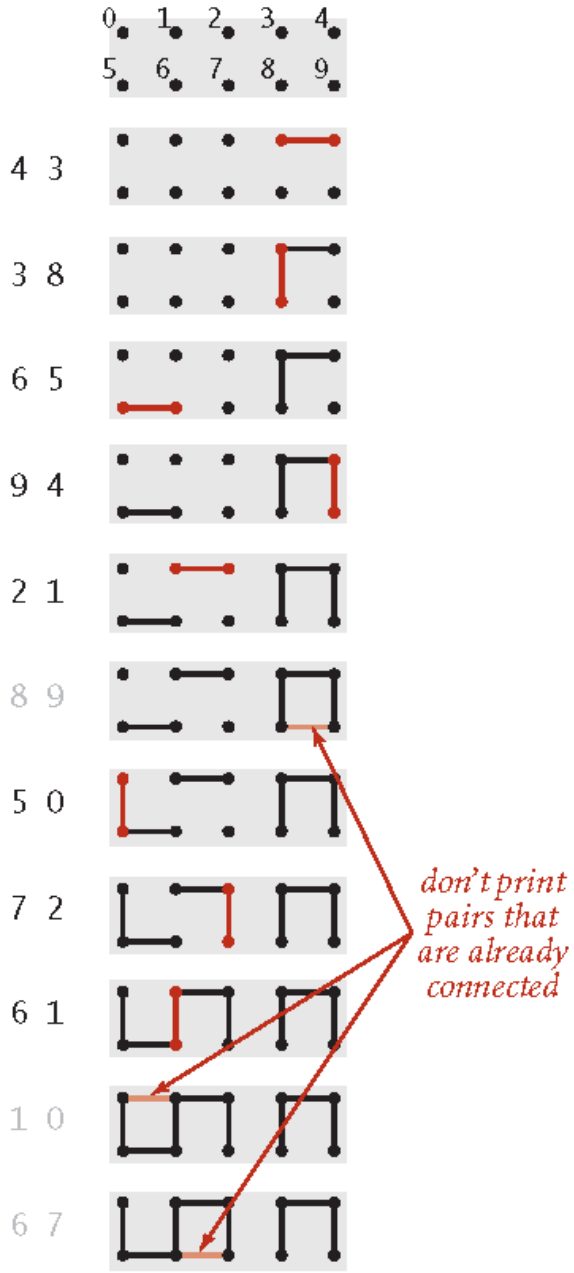
- Minim Local
  - Se dă un vector de elemente distincte arr.
  - Se cere să se determine un element arr[i] care e mai mic decât vecini săi (1 vecin dacă arr[i] e la unul din capetele vectorului, 2 vecini dacă arr[i] este în interiorul vectorului)
- MinimLocal.cs
- Hint: http://stackoverflow.com/questions/12238241/find-local-minima-in-an-array
- http://courses.csail.mit.edu/6.006/spring11/lectures/lec02.pdf

Analiza algoritmilor
Studiu de caz – Union Find

- Conectivitate dinamică:
  - Datele de intrare sunt perechi de numere p și q
  - Perechea p, q o interpretăm „p este conectat cu q”
  - „este conectat” este o relație de echivalență
    - Reflexivă – p este conectat cu p
    - Simetrică – dacă p este conectat cu q atunci și q este conectat cu p
    - Tranzitivă – daca p este conectat cu q și q este conectat cu r atunci p este conectat cu r
  - O relație de echivalență partiționează mulțimea în clase de echivalență
  - Două obiecte sunt în aceeași clasă de echivalență dacă sunt conectate

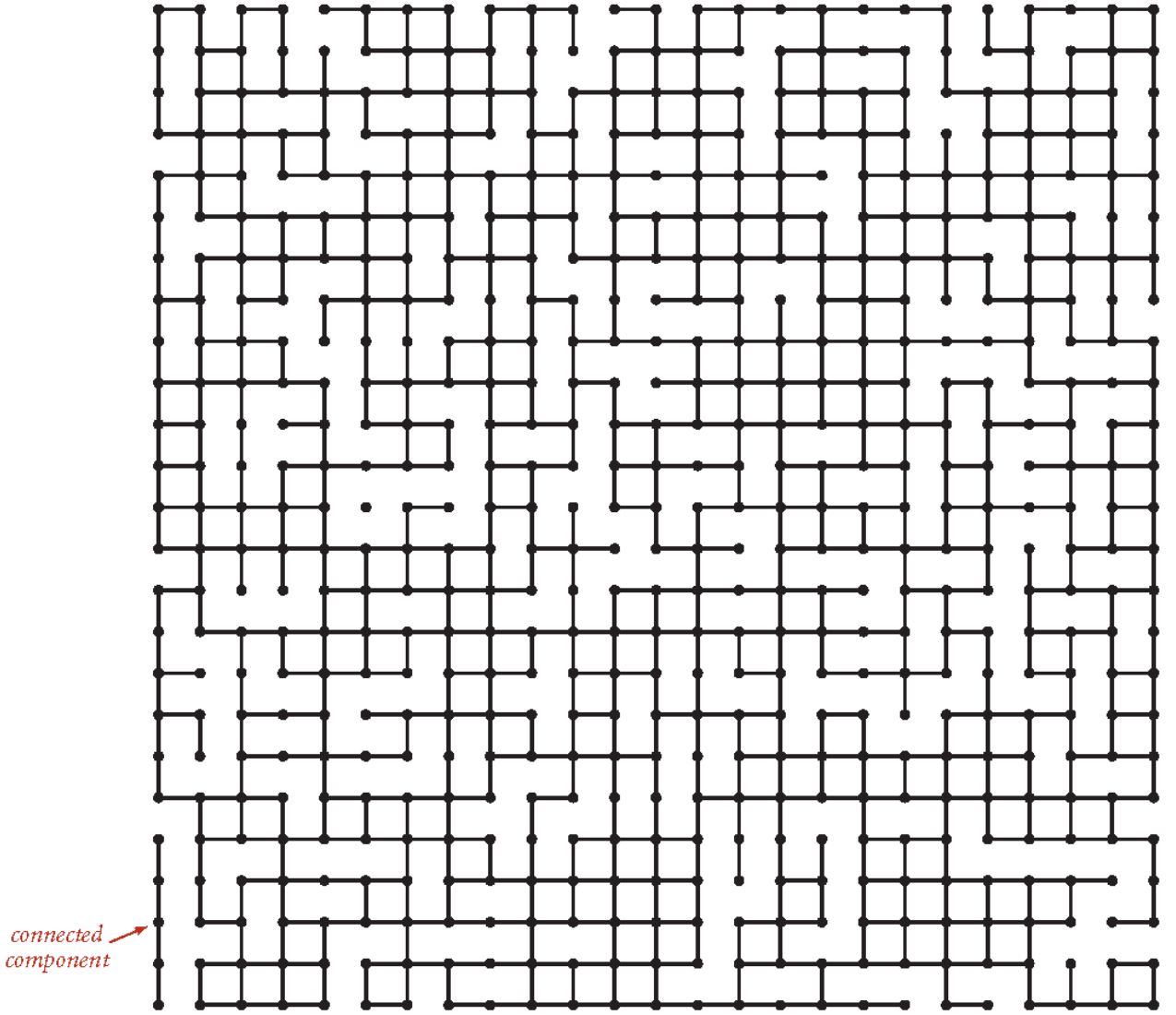
Analiza algoritmilor
Studiu de caz – Union Find

- Mulțimea e formată din zece elemente
- Urmează o lista de conexiuni
- Dacă pentru o pereche (p, q) din intrare se stabilește că elementele sunt deja conectate atunci perechea este ignorată
- Aplicații:
  - Rețele de calculatoare (calculatoare, conexiuni între ele)
  - Rețele sociale (punctele sunt persoane, legăturile sunt relații de prietenie



Analiza algoritmilor
Studiu de caz – Union Find

- Un exemplu 25 x 25 cu 900 de conexiuni si 3 componente conexe
- Se pot „vedea” cele 3 componente?



Medium connectivity example (625 sites, 900 edges, 3 connected components)

Analiza algoritmilor
Studiu de caz – Union Find

- API pentru Union Find

```
public UF(int n) // Initializare
public void union(int p, int q) // Stabileste o conexiune intre p si q
public int find(int p) // Determina componenta in care se afla p
public bool connected(int p, int q) // Determina daca p si q sunt conectate
public int count() // Determina numarul de componente conexe
```

Analiza algoritmilor
Studiu de caz – Union Find

- Quick Find
  - p și q sunt conectate dacă și numai dacă id[p] == id[q]

```
public int find(int p)
        {
            return id[p];
        }
public bool connected(int p, int q)
        {
            return find(p) == find(q);
        }
```

Analiza algoritmilor
Studiu de caz – Union Find

- Quick Find
  - p și q sunt conectate dacă și numai dacă id[p] == id[q]

```
public void union(int p, int q)
        {
            int pId = find(p), qId = find(q);
            if (pId == qId)   return;
            for (int i = 0; i < id.Length; i++)
                if (id[i] == pId)
                    id[i] = qId;
            componentNo--;
        }
```

Analiza algoritmilor
Studiu de caz – Union Find

- Quick Find
  - Operația find() este foarte rapidă – un singur acces la elementele vectorului
  - Operația union() este ineficientă – trebuie să parcurgă toate elementele vectorului O (n)
- Propoziție: algoritmul QuickFind folosește un acces la elementele vectorului pentru operția find() și între N+3 și 2N+1 accese la elementele vectorului pentru operația union()
- Complexitatea algorimului QuickFind ~N^2
- Algoritmul este nefezabil atunci când N este foarte mare

Analiza algoritmilor
Studiu de caz – Union Find

- Quick Union
  - Folosim același vector id[] dar interpretăm altfel valorile – „vector de tați”
  - id[i] = părintele lui i într-un arbore
  - Dacă id[i] == i atunci i este nod rădăcină
  - p și q sunt conectate dacă fac parte din același arbore

Analiza algoritmilor
Studiu de caz – Union Find

- Quick Union

```
public int find(int p)
        {
            while (p != id[p])
                p = id[p];

            return p;
        }
```

Analiza algoritmilor
Studiu de caz – Union Find

- Quick Union

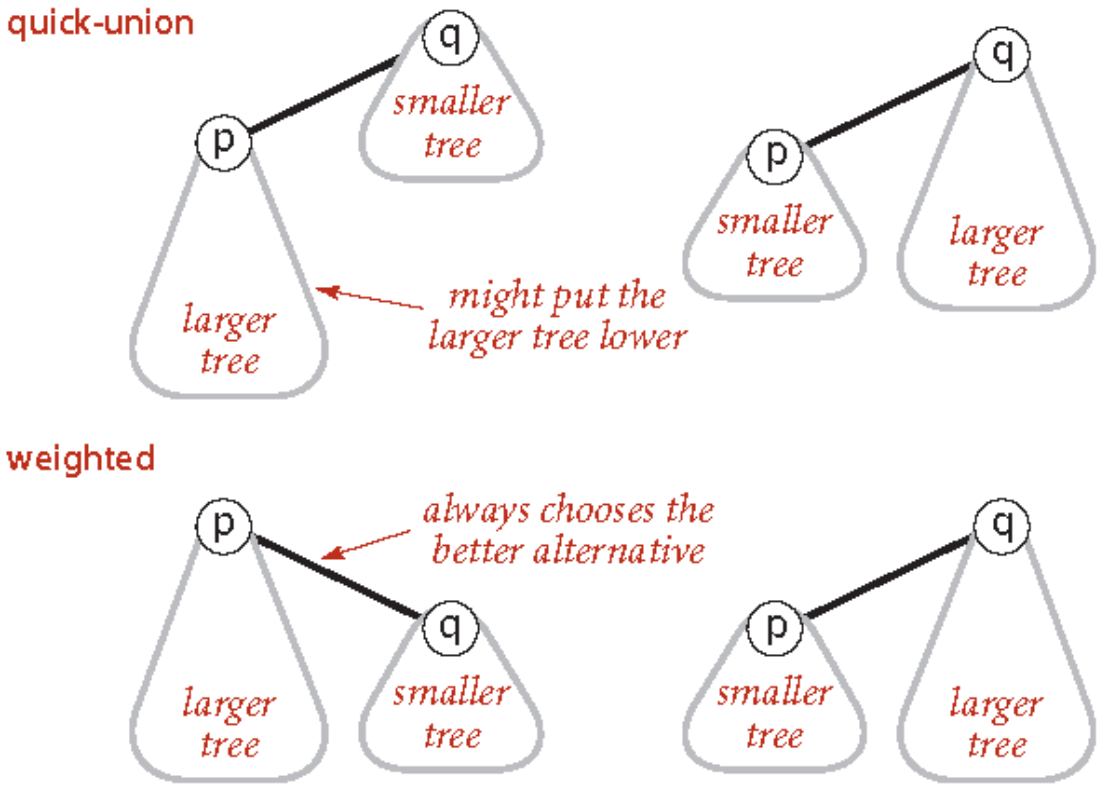
```
public void union(int p, int q)
        {
            int pRoot = find(p), qRoot = find(q);
            if (pRoot != qRoot)
            {
                id[pRoot] = qRoot;
                componentNo--; // numarul de componente scade cu 1
            }
        }
```

Analiza algoritmilor
Studiu de caz – Union Find

- Quick Union
  - este o ușoară îmbunătățire față de QuickFind deoarece pentru anumite date de intrare timpul de execuție este liniar
  - nu putem garanta eficiența algoritmului
- Nu ne va rezolva în mod eficient problema atunci când numărul de date din intrare este mare

Analiza algoritmilor
Studiu de caz – Union Find

- Weighted Quick Union



Analiza algoritmilor
Studiu de caz – Union Find

- Weighted Quick Union
  - Garantează performanță logaritmică
- Propoziție: Înălțimea oricărui arbore dintr-o pădure cu N noduri nu depășește log2 N
  - Demonstrație: prin inducție demonstrăm că înălțimea unui arbore de dimensiune k este cel mult log2 k
  - Pentru k = 1 înălțimea arborelui este 0
  - Ipoteza de inducție: un arbore cu i noduri are cel mult înălțimea log2 i, pt i < k
  - Dacă trebuie să combinăm un arbore de dimensiune i cu un arbore de dimensiune j (i <= j, i + j = k) înălțimea fiecărui nod din arborele mai mic crește cu 1 dar aceste noduri vor fi într-un arbore de dimensiune k
  - 1 + log2 i = log2 (i + i) <= log2 (i + j) = log2 k

Sortare

- Sortare = procesul de rearanjare a elementelor dintr-o secvență astfel încât să fie într-o anumită ordine logică
- Sortarea este de regulă prima operație în organizarea datelor
- Orice sistem de calcul are implementări pentru algoritmi de sortare atât pentru a fi utilizați de sistem cât și de utilizatori
- Motive practice pentru studiul algoritmilor de sortare
  - Reprezintă o introducere în modalitățile folosite pentru compararea eficienței algoritmilor
  - Tehnici asemănătoare pot fi folosite pentru a rezolva alte probleme
  - Sortarea este folosită de multe ori ca un prim pas în rezolvarea unei probleme

Sortare

- Algoritmii de sortare se folosesc în foarte multe domenii pentru a ordona diverse entități
- Quicksort – este considerat unul din cei mai importanți 3 algoritmi din știință și inginerie ai sec. XX
- Vom discuta:
  - Metode de sortare clasice
  - Implementare eficientă pentru un tip de date fundamental: *coadă cu prioritate*
  - Baze teoretice pentru compararea algoritmilor de sortare

Sortare

- Vom rearanja vectori de elemente – unde fiecare element conține o cheie de sortare
- După sortare cheile vor fi într-o anumită ordine bine definită (de regulă numerică sau alfabetică)
- Vectorul va fi rearanjat a.î.
  - Cheia fiecărui element nu va fi mai mică decât cheia unui element cu index mai mic
  - Cheia fiecărui element nu va fi mai mare decât cheia unui element cu index mai mare
- Caracteristicile cheii și ale elementelor pot varia foarte mult

Sortare

- Toți algoritmii de sortare pe care îi vom trata vor fi prezentați în contextul unei clase la fel ca și în ExampleSort.cs
- Folosim interfața Icomparable
- Clasa pentru implementarea unui algoritm de sortare va conține următoarele metode

```
public static void Main(string[] args)
public static void sort<T>(T[] a) where T: IComparable<T>
private static bool less<T>(T p, T q) where T: IComparable<T>
private static void exch<T>(T[] a, int i, int j) where T: IComparable<T>
private static void show<T>(T[] a) where T: IComparable<T>
public static bool isSorted<T>(T[] a) where T: IComparable<T>
```

Sortare

- Fiecare algoritm de sortare accesază datele doar cu metodele less() și exch()
- Aceste metode sunt foarte ușor de implementat datorită interfeței IComparable
- Restricționarea accesului la date doar pentru aceste două metode ne oferă o serie de avantaje
  - Codul devine mai ușor de citit
  - Codul este portabil
  - E mai ușor de verificat corectitudinea algoritmului
  - Mai ușor de studiat performanța algoritmului
  - Mai ușor de comparat algoritmii

Sortare

- Certificare
  - Verificarea faptului că algoritmul ordonează vectorul indiferent de elementele pe care le conține
  - Am inclus o aserțiune în cod care verifică acest lucru `Debug.Assert(isSorted(a), "Vectorul nu este sortat");`
  - Aserțiunea este suficientă doar dacă folosim doar metoda exch()
  - Dacă folosim și alte metode care modifică elementele vectorului inițial aserțiunea poate fi corectă fără ca algoritmul de sortare să fie corect

Sortare

- Timpul de execuție
  - Vom testa performanța algoritmilor
  - Ne interesează numărul de operații elementare (comparații, interschimbări sau numărul de accesări la elementele vectorului pentru citire/scriere)
  - Modelul de cost pentru sortare e reprezentat de numărul de comparații și interschimbări. Dacă algoritmul nu face interschimbări vom contoriza numărul de accesări ale elementelor vectorului
  - Pe baza acestor elemente vom emite ipoteze în legătură cu performanța algoritmilor și vom verifica experimental aceste ipoteze

Sortare

- Memorie suplimentară
  - Memoria folosită de un algoritm de sortare este un factor la fel de important ca și timpul de execuție
  - Există două tipuri de algoritmi: *in-place* (nu au nevoie de memorie suplimentară, cu excepția unui număr constant de valori) și algoritmi care au nevoie de memorie suplimentară a cărei dimensiune este egală cu dimensiunea datelor ce sunt sortate.
- Tipuri de date
  - Codul pe care l-am scris va putea sorta orice valori care sunt instanțe ale unui tip de date ce implementează interfața *IComparable*.
  - Metoda *CompareTo()* trebuie să implementeze o relație de ordine totală (reflexivă, antisimetrică (dacă v < w at. w > v și dacă v = w atunci w = v) și tranzitivă)
  - Exemplu: Date.cs

Sortare
Selection Sort

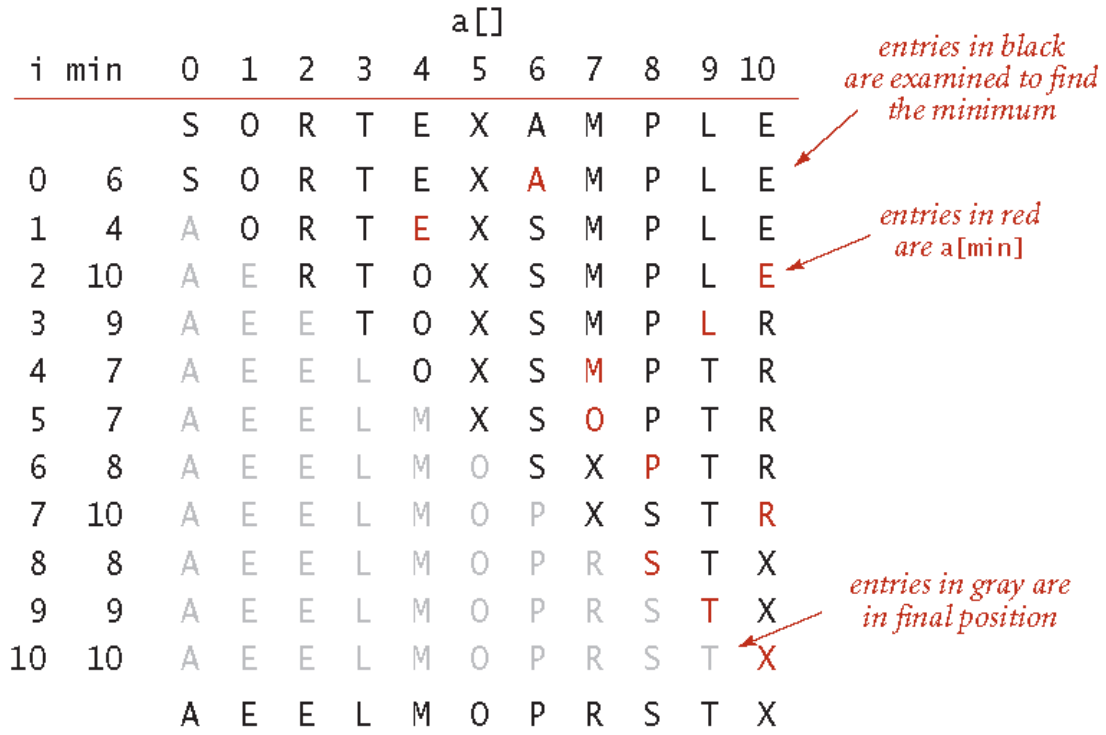
- Un algoritm foarte simplu
  - Găsim cel mai mic element din vector [0, n - 1] și îl interschimbăm cu elementul de pe prima poziție
  - Găsim cel mai mic element din vector [1, n - 1] și îl interschimbăm cu elementul de pe a doua poziție
  - Continuăm până la ultimul element
  - Selectăm în mod repetat cel mai mic element din cele rămase
- http://www.sorting-algorithms.com/selection-sort

Sortare
Selection Sort

```
public static void sort<T>(T[] a) where T: IComparable<T> {
            int i, j, n = a.Length, min;
            for (i = 0; i < n; i++)
            {
                min = i;
                for (j = i + 1; j < n j++)
                    if (less(a[j], a[min]))
                        min = j;
                exch(a, i, min); // se executa de n ori
            }
        }
```

Sortare
Selection Sort

- **Propoziție:** Selection sort folosește ~N^2 / 2 comparații și N interschimbări pentru a sorta un vector de lungime N



Sortare
Selection Sort

- Timpul de execuție nu depinde de datele de intrare
- Indiferent de ordinea datelor din intrare timpul de execuție este același
- Dacă vectorul este sortat de la început sau dacă toate elementele din vector sunt egale timpul de execuție rămâne ~N^2
- Se fac N interschimbări – un număr mic în comparație cu alți algoritmi de sortare

Sortare
Insertion Sort

- Similar cu sortarea cărților de joc
- Fiecare element este inserat la locul potrivit între cele care au fost deja tratate
- Algoritmul trebuie să facă loc pentru inserare prin mutarea elementelor înspre dreapta
- Elementele din stânga indexului curent sunt sortate dar nu sunt în mod obligatoriu în poziția finală.
- În dreapta indexului curent pot fi elemente mai mici decât cele tratate deja
- Când indexul ajunge la ultimul element vectorul este sortat.
- http://www.sorting-algorithms.com/insertion-sort

Sortare
Insertion Sort

```
public static void sort<T>(T[] a) where T: IComparable<T>
{
            int i, j, n = a.Length;
            for (i = 1; i < n; i++)
            {
                for (j = i; j > 0 && less(a[j], a[j - 1]); j--)
                    exch(a, j, j - 1);
            }
}
```

Sortare
Insertion Sort

- **Propoziție:** Insertion sort folosește în medie ~N^2 / 4 comparații și ~N^2 / 4 interschimbări pentru a sorta un vector de lungime N cu chei distincte, aleatoare. În cel mai rău caz avem ~N^2 / 2 comparații și ~N^2 / 2 interschimbări, iar în cel mai bun caz N – 1 comparații și 0 interschimbări

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

entry in red is a[j]

entries in black moved one position right for insertion

Sortare
Insertion Sort

- Timpul de execuție depinde de ordinea inițială a elementelor în vector
- Dacă vectorul este mare și este deja sortat (sau aproape sortat) *InsertionSort* va fi mult mai rapid decât *SelectionSort*.
- Pentru un vector care este deja sortat timpul de execuție al *InsertionSort* este liniar (în acest caz timpul de execuție al *SelectionSort* este pătratic)
- Când numărul de inversiuni din vector este mic (vectorul este aproape sortat) *InsertionSort* poate fi mai rapid decât orice alt algoritm de sortare (inclusiv QuickSort)
- Numărul de accese la elementele vectorului poate fi înjumătățit prin mutarea elementelor mai mari decât a[i] cu o poziție la dreapta și efectuarea unei interschimbări la final când este găsită poziția în care trebuie inserat a[i] (InsertionXSort.cs)

Sortare
Compararea algoritmilor de sortare

- Compararea algoritmilor se face prin metoda științifică:
  - Implementarea și depanarea algoritmilor
  - Analizarea proprietăților lor de bază
  - Formularea unor ipoteze în legătură cu performanța comparativă
  - Rularea unor experimente pentru a valida ipotezele (SortCompare.cs)
- **Propoziție:** timpul de rulare pentru InsertionSort și SelectionSort este pătratic. InsertionXSort este de aproximativ două ori mai rapid decât SelectionSort.
- InsertionXSort este o optimizare a algoritmului InsertionSort. Numărul de interschimbări este înjumătățit. (InsertionXSort.cs)

Sortare ShellSort

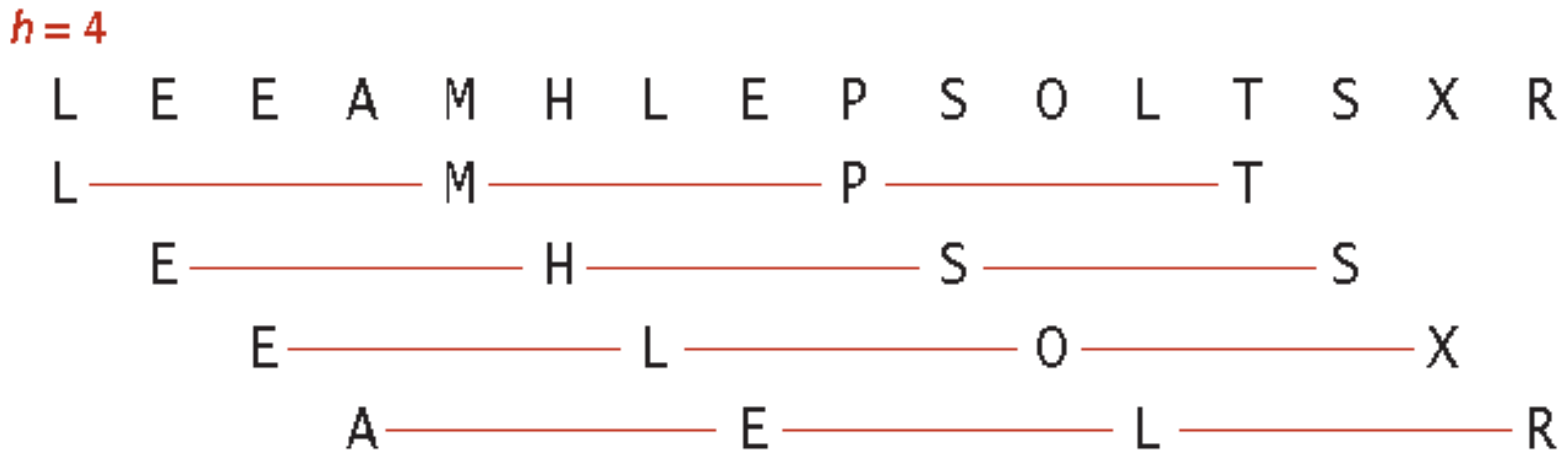
- *ShellSort* este un algoritm de sortare rapid bazat pe *InsertionSort*
- *InsertionSort* interschimbă întotdeauna două elemente care sunt unul lângă altul în vector. Dacă cel mai mic element al vectorului este pe ultima poziție vor fi nevoie de N-1 operații de interschimbare pentru a fi adus pe prima poziție (ceea ce este ineficient)
- *ShellSort* este o simplă extensie mai rapidă a *InsertionSort* întrucât permite interschimbarea a două elemente care sunt mai îndepărtate unul de altul rezultând vectori parțial sortați care pot fi sortați în mod eficient (prin *InsertionSort*)

Sortare ShellSort

- Ideea este de a rearanja vectorul a.î. luând tot al *h*-lea element din vector obținem o secvență sortată.
- Un astfel de vector se numește *h*-sortat
- Un vector *h*-sortat reprezintă o intercalare a h secvențe sortate
- Prin *h*-sortare pentru o valoarea mare a lui *h*, putem muta elemente pe distanțe mari astfel încât va fi mai ușor să *h*-sortăm pentru valori mici ale lui *h*
- O astfel de procedură pentru orice secvență de valori pentru h care se termină cu 1 va produce un vector sortat: *shellsort*

Sortare ShellSort

- O secvență *h*-sortată reprezintă h subsecvențe sortate, intercalate.



Sortare
ShellSort

```
public static void sort<T>(T[] a) where T : IComparable<T>  {
  int n = a.Length;
  int h = 1; // 3x+1 secv. de incrementuri:  1, 4, 13, 40, 121, 364, 1093, ...
   while (h < n / 3) h = 3 * h + 1;
   while (h >= 1) {
        for (int i = h; i < n; i++)  // h-sortare a vectorului
           for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
        h /= 3;
   }
}
```

Sortare ShellSort

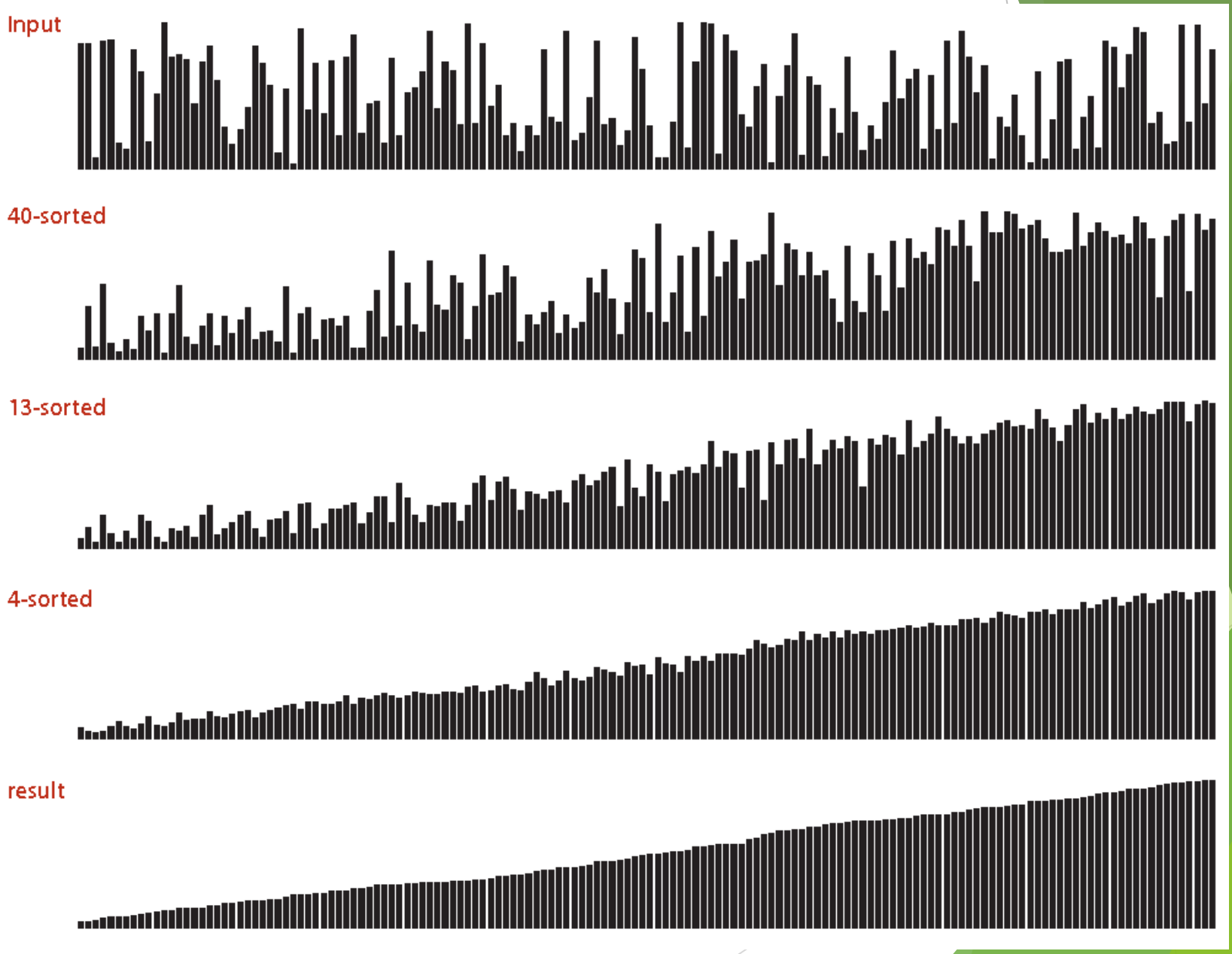
- Implementarea folosește secvența de valori descrescătoare ½(3^k - 1) care începe la cea mai mare valoarea din această secvență mai mică decât N/3 și se termină cu 1 // `1, 4, 13, 40, 121, 364, 1093, ...`
- Se pot folosi și alte secvențe de incremenți cu performanțe mai bune. Ne rezumăm la aceasta întrucât e simplu de calculat
- ShellSort are eficiență superioară făcând compromis între dimensiunea vectorului și ordinea parțială din subsecvențe
- Înțelegerea performanței ShellSort este dificilă – fiind un algoritm de sortare pentru care performanța în cazul unor vectori de elemente aleatoare nu a fost caracterizată în mod precis în literatura de specialitate
- http://www.sorting-algorithms.com/shell-sort

Sortare ShellSort

- *ShellSort* este mult mai rapid decât *InsertionSort* și *SelectionSort*
- Avantajele *ShellSort* cresc pe măsură ce crește dimensiunea vectorului
- *ShellSort* permite sortarea unor secvențe de dimensiune mare care nu ar putea fi sortate în timp util cu celelalte două metode
- **Principiu în algoritmică:** *motivul principal pentru care studiem performanța și proiectarea algoritmilor este de a obține algoritmi mai buni care permit găsirea soluțiilor pentru diverse probleme*
- Numărul de comparații efectuat de *ShellSort* în cel mai rău caz este N^(3/2)

Sortare ShellSort

- Vizualizare ShellSort



Sortare
Merge Sort

- Algoritmul MergeSort se bazează pe o operație numită *interclasare* (merge)
- Interclasare = combinarea a doi vectori sortați pentru a forma un nou vector sortat (care conține toate elementele din cei doi vectori)
- MergeSort = algoritm recursiv – se împarte vectorul în două jumătăți, se sortează cele două jumătăți, se combină (interclasează) cele două jumătăți sortate pentru a obține vectorul original sortat. (Un vector cu un singur element este sortat)
- Avantaj: timpul de execuție este N log N (N = numărul de elemente din vector)
- Dezavantaj: se folosește spațiu suplimentar proporțional cu N

Sortare
Merge Sort

- Operația Merge (interclasare)

```
private static void merge<T>(T[] a, int lo, int mid, int hi, T[] aux) where T:
IComparable<T>
        {
            int i = lo, j = mid + 1;
            for (int k = lo; k <= hi; k++) // copiem portiunea din a in aux
                aux[k] = a[k];
            for (int k = lo; k <= hi; k++)
                if (i > mid)   a[k] = aux[j++];
                else if (j > hi)   a[k] = aux[i++];
                else if (less(aux[j], aux[i]))  a[k] = aux[j++];
                else   a[k] = aux[i++];
        }
```

Sortare
Merge Sort

- Metode in-place de realizare a operației merge sunt foarte complicate
- Folosim o metodă care folosește un singur vector auxiliar de dimensiune N pe care îl alocăm **o singură** dată și îl transmitem ca argument la fiecare apel

		a[]												aux[]									
	k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9
input		E	E	G	M	R	A	C	E	R	T			–	–	–	–	–	–	–	–	–	–
copy		E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	A	C	E	R	T
												0	5										
	0	A										0	6	E	E	G	M	R	A	C	E	R	T
	1	A	C									0	7	E	E	G	M	R		C	E	R	T
	2	A	C	E								1	7	E	E	G	M	R			E	R	T
	3	A	C	E	E							2	7		E	G	M	R			E	R	T
	4	A	C	E	E	E						2	8			G	M	R			E	R	T
	5	A	C	E	E	E	G					3	8			G	M	R				R	T
	6	A	C	E	E	E	G	M				4	8				M	R				R	T
	7	A	C	E	E	E	G	M	R			5	8					R				R	T
	8	A	C	E	E	E	G	M	R	R		5	9									R	T
	9	A	C	E	E	E	G	M	R	R	T	6	10										T
merged result		A	C	E	E	E	G	M	R	R	T												

Sortare
Merge Sort

- Top-Down mergesort – implementare recursivă a algoritmului MergeSort bazată pe operația de interclasare
- Este unul din cele mai cunoscute exemple ale paradigmei *divide-et-impera* pentru realizarea de algoritmi eficienți (de realizat un *call trace*!!)

```
private static void sort<T>(T[] a, int lo, int hi, T[] aux) where T:
IComparable<T>
        {
            if (hi <= lo)  return;
            int mid = lo + (hi - lo) / 2;
            sort(a, lo, mid, aux);
            sort(a, mid + 1, hi, aux);
            merge(a, lo, mid, hi, aux);
        }
```

Sortare
Merge Sort

Call trace (apelurile pentru vector de dimensiune 1 sunt omise)

```
sort(a, 0, 15)
sort left half  sort(a, 0, 7)
    sort(a, 0, 3)
      sort(a, 0, 1)
        merge(a, 0, 0, 1)
      sort(a, 2, 3)
        merge(a, 2, 2, 3)
      merge(a, 0, 1, 3)
    sort(a, 4, 7)
      sort(a, 4, 5)
        merge(a, 4, 4, 5)
      sort(a, 6, 7)
        merge(a, 6, 6, 7)
      merge(a, 4, 5, 7)
    merge(a, 0, 3, 7)
```

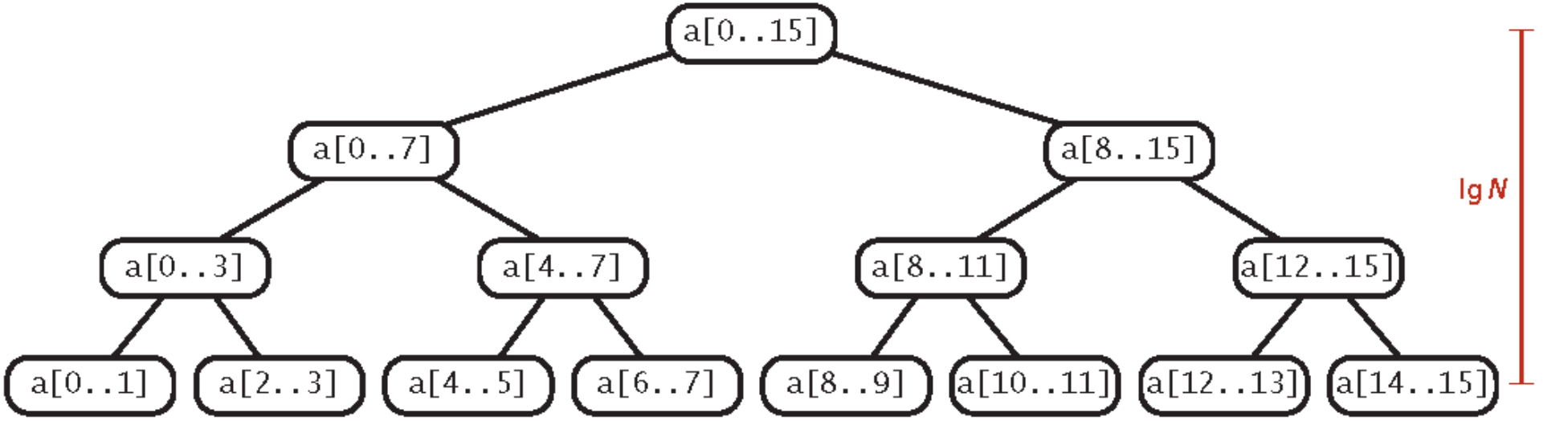
```
sort right half  sort(a, 8, 15)
      sort(a, 8, 11)
        sort(a, 8, 9)
          merge(a, 8, 8, 9)
        sort(a, 10, 11)
          merge(a, 10, 10, 11)
        merge(a, 8, 9, 11)
      sort(a, 12, 15)
        sort(a, 12, 13)
          merge(a, 12, 12, 13)
        sort(a, 14, 15)
          merge(a, 14, 14,15)
        merge(a, 12, 13, 15)
    merge(a, 8, 11, 15)
merge results  merge(a, 0, 7, 15)
```

Sortare
Merge Sort

- **Propoziție:** Top-down MergeSort folosește între ½ N log N și N log N operații de comparație pentru a sorta un vector de lungime N
- **Demonstrație:**
  - fie C(N) numărul de operații de comparație necesar pentru sortarea unui vector de lungime N
  - C(0) = C(1) = 0
  - C(N) ≤ C(N / 2) + C(N / 2) + N (stânga + dreapta + merge)
  - C(N) ≥ C(N / 2) + C(N / 2) + N / 2 (pentru că numărul de comparații pentru merge este cel puțin N / 2
  - Dacă N = 2^n atunci C(2^n) = 2C(2^(n-1)) + 2^n
  - C(2^n)/2^n = C(2^(n-1)) / 2^(n-1) + 1 = C(2^(n-2))/ 2^(n-2) + 1 + 1 =…=C(2^0)/2^0 + n = n
  - C(N) = C(2^n) = n * 2^n = N * log N

Sortare
Merge Sort

- Un alt mod de a înțelege propoziția anterioară



Sortare
Merge Sort

- Fiecare nod din arbore reprezintă un subvector pentru care sort() face o interclasare
- Arborele are n niveluri
- Pentru k = 0, n – 1 fiecare nivel k are 2^k subvectori de lungime 2^(n - k)
- Orice astfel de subvector necesită 2^(n - k) operații de comparare pentru realizarea interclasării
- Vom avea 2^k * 2^(n - k) = 2^n comparații pentru fiecare nivel
- În total numărul de comparații va fi n*2^n = N log N

Sortare
Merge Sort

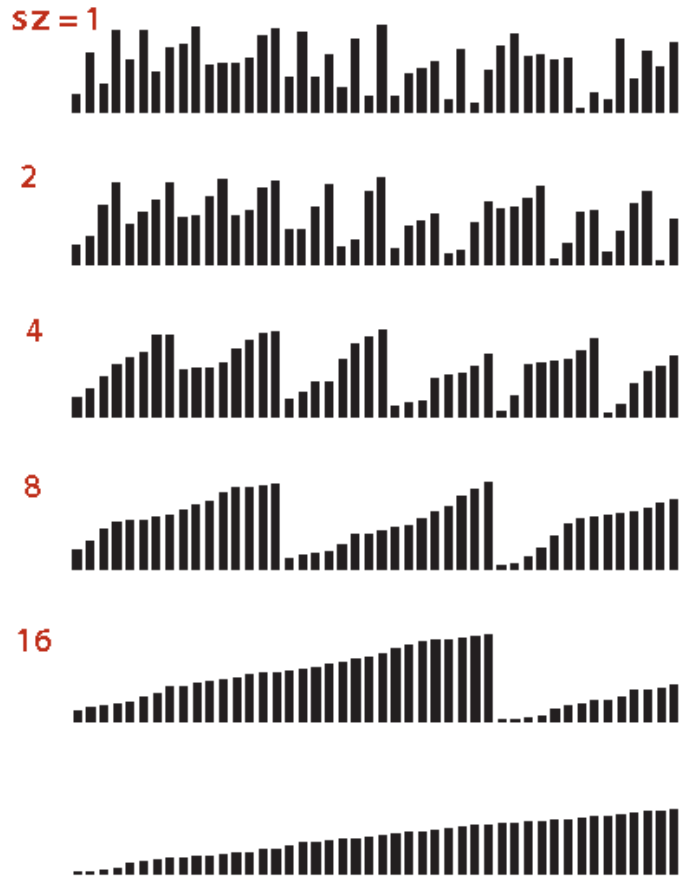
- **Propoziție:** Top-down MergeSort folosește cel mult 6N log N accese la elementele vectorului pentru a-l sorta.
- **Demonstrație:**
  - fiecare operație merge() folosește cel mult 6N accese (2N pentru copiere în aux, 2N pentru a fi mutate înapoi și cel mult 2N pentru comparații)
- Din cele două propoziții rezultă că timpul necesar pentru ca MergeSort să sorteze vectorul este N log N
- Prin urmare MergeSort ne permite abordarea cu succes a unor probleme de sortare a unor vectori mult mai mari, care nu ar putea fi sortați în timp util de algoritmii de sortare elementari

Sortare
Merge Sort

- Cu mici modificări putem îmbunătăți performanța:
  - Vectorii de dimensiune mică (de ex. ≤ 15) pot fi sortați cu InsertionSort pentru că performanța va fi mai bună (de verificat!!!). Timpul de execuție trebuie să se îmbunătățească cu 10-15%
  - Asta e o tehnică generală ce poate fi aplicată la orice algoritm recursiv (o instanță de dimensiune mică a problemei se rezolvă nerecursiv)
  - Testăm dacă vectorul este deja sortat – dacă a[mid] ≤ a[mid + 1] nu mai apelăm merge()
  - Eliminarea copierii în vectorul auxiliar; putem elimina timpul de copiere în vectorul auxiliar dar nu și spațiul: pentru cele două invocări ale metodei sort() una va lua intrare din a și pune vectorul sortat în aux, iar cealaltă va lua intrarea din aux și o pune vectorul sortat în a. La fiecare apel recursiv schimbăm rolul lui a cu aux și invers.
- http://www.sorting-algorithms.com/merge-sort

Sortare
Merge Sort - BottomUp

- MergeSort se poate implementa și nerecursiv
  - La prima parcurgere a vectorului executăm merge() pe fiecare pereche de elemente rezultând perechi ordonate
  - La următoarea parcurgere executăm merge() pe vectori de lungime 2 rezultând vectori ordonați de lungime 4 etc.



Sortare
Merge Sort - BottomUp

```
public static void sort<T>(T[] a) where T : IComparable<T>
{
    T[] aux = new T[a.Length];
    int n = a.Length;
    for (int sz = 1; sz < n; sz = sz + sz)
      for (int lo = 0; lo < n - sz; lo += sz + sz)
        merge(a, lo, lo + sz - 1, Math.Min(lo + sz + sz - 1, n - 1), aux);
}
```

Sortare
Merge Sort - BottomUp

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Sortare
Merge Sort - BottomUp

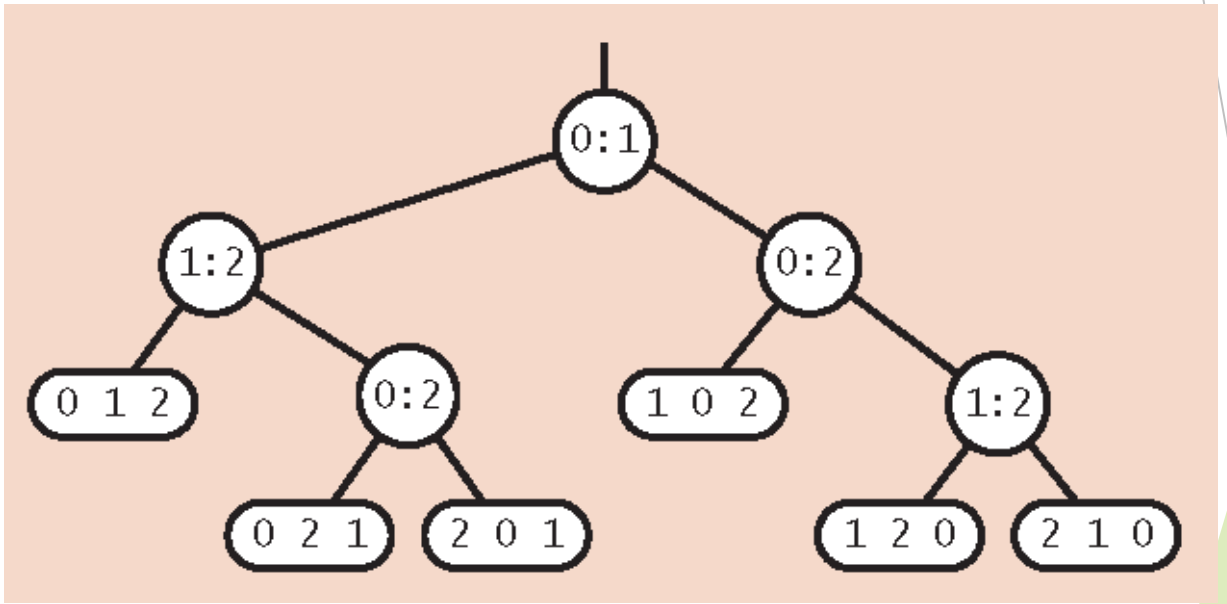
- **Propoziție:** BottomUpMergeSort efectuează între ½ N log N și N log N comparații și cel mult 6N log N accesări ale elementelor vectorului
- **Demonstrație:**
  - Numărul de parcurgeri ale vectorului este [log N] (valoarea lui n pentru care 2^n ≤N < 2^(n+1))
  - La fiecare parcurgere numărul de accesări ale elementelor vectorului este exact 6N iar numărul de comparații este cel mult N dar nu mai puțin de N/2
- Pentru sortarea unei liste înlănțuite se folosește o versiune a BottomUpMergeSort – se rearanjează legăturile pentru a sorta lista *in-place*

Sortare
Complexitatea sortării

- MergeSort este important întrucât ne permite să demonstrăm un rezultat fundamental de teroria complexității calculului care ne ajută să înțelegem cât de dificilă este operația de sortare
- Modelul de calcul va fi bazat pe operațiile de comparare (accesurile la elementele vectorului vor fi ignorate)
- **Propoziție:** nici un algoritm de sortare bazat pe comparații nu poate sorta N elemente cu mai puțin de log(N!) ~ N log N comparații.
- **Demonstrație:** pp. că avem chei distincte.
  - Cu ajutorul unui arbore binar vom descrie secvența de comparații.
  - Fiecare nod va fi fie frunză ce indică faptul că sortarea s-a încheiat și s-a stabilit ordinea elementelor
  - Fie un nod intern (i:j) care corespunde operației de comparare între a[i] și a[j].
  - Subarborele stâng corespunde situației a[i] < a[j] iar subarborele drept a[i] > a[j]

Sortare
Complexitatea sortării

- Fiecare drum de la rădăcină la o frunză corespunde unei secvențe de operații de comparare pe care algoritmul o folosește pentru a stabili ordinea din acea frunză

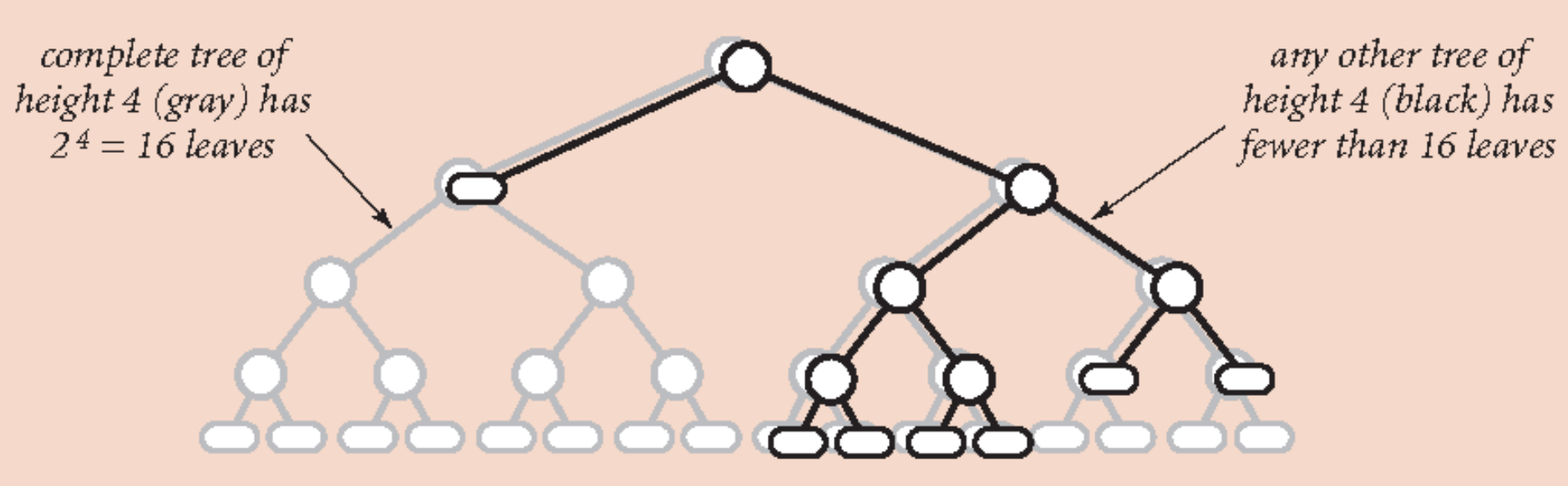


Sortare
Complexitatea sortării

- Arborele trebuie să aibă N! frunze pentru că sunt N! permutări de N chei distincte
- Dacă arborele nu are N! frunze atunci înseamnă că lipsesc anumite permutări (ordonări ale cheilor) pentru care algoritmul de sortare nu va funcționa corect
- Ne interesează *înălțimea* arborelui = lungimea celui mai lung drum de la rădăcină la o frunză.
- Înălțimea arborelui reprezintă numărul de operații de comparație pe care le face algoritmul în cel mai rău caz
- Obs. Un arbore de înălțime *h* nu poate avea mai mult de *2^h* frunze. Arborele de înălțime h cu numărul maxim de frunze este arborele complet (perfect echilibart) – exemplu pentru h = 4 pe slide-ul următor

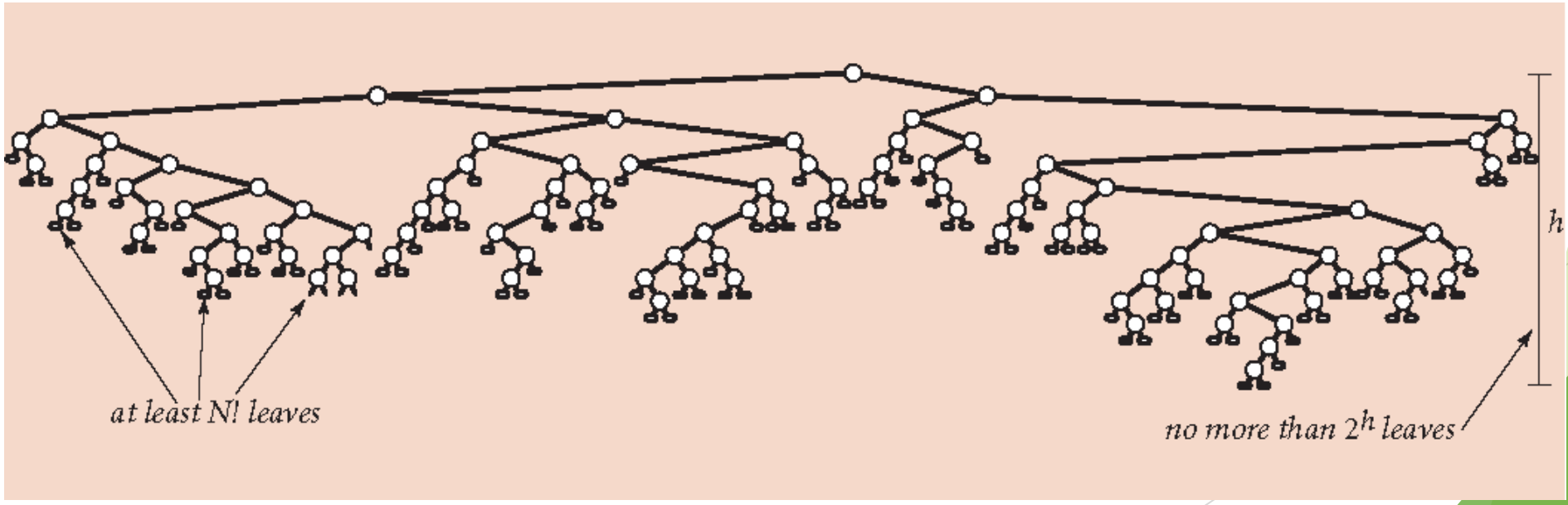
Sortare
Complexitatea sortării

- Arbore de înălțime 4 complet (perfect echilibrat) = 16 frunze
- Orice alt arbore de înălțime 4 are mai puțin de 16 frunze



Sortare
Complexitatea sortării

- Orice algoritm de sortare bazat pe comparații corespunde unui arbore de comparații de înălțime h pentru care *N! ≤ numărul de frunze ≤ 2^h*
- *h* reprezintă numărul de comparații în cel mai rău caz
- Logaritmând în baza 2 obținem că numărul de comparații - *h* - trebuie să fie minim log2(N!) ~ N log N (pe baza aproximării lui Stirling)



Sortare
Complexitatea sortării

- Numărul de comparații folosit de MergeSort (în cel mai rău caz) este ~N log N
- Această valoarea este o limită superioară pentru dificultatea problemei sortării
- Nici un algoritm de sortare bazat pe comparații nu poate sorta cu mai puțin de ~N log N comparații – această fiind o limită inferioară
- **Propoziție:** MergeSort este un algoritm de sortare bazat pe comprații optim din punct de vedere asimptotic
- Vom considera și alte metode de sortare pentru că:
  - MergeSort nu este optim dpdv al spațiului folosit
  - Și alte operații pe lângă comparații pot fi importante (accesele la elementele vectorului)
  - Anumite date pot fi sortate fără comparații

Sortare QuickSort

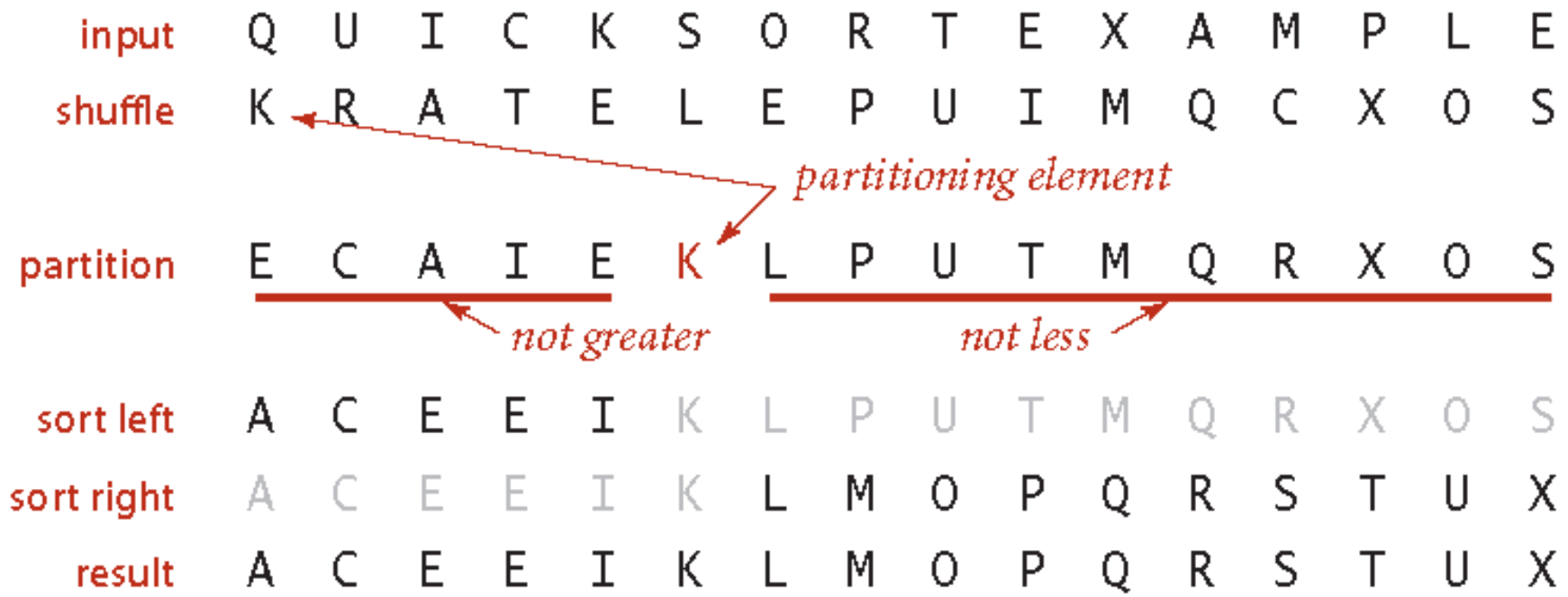
- QuickSort este unul din cei mai utilizați algoritmi de sortare
- Inventat în 1960 de C.A.R. Hoare
- Are o serie de avantaje:
  - Ușor de implementat
  - Funcționează bine pentru multe tipuri de date
  - Substanțial mai rapid față de orice altă metodă de sortare în aplicații obișnuite
  - In-place (folosește un număr constant de variabile suplimentare)
  - Timpul de execuție este proporțional cu N log N pentru un vector de lungime N
  - Este rapid atât în teorie cât și în practică

Sortare QuickSort

- QuickSort este un algoritm de sortare de tip divide-et-impera
- Ideea principală este de a partiționa vectorul în doi sub-vectori (stâng și drept) care vor fi sortați independent (în urma partiționării toate elementele din primul subvector vor fi mai mici decât elementele din cel de-al doilea subvector)
- QuickSort este complementar lui MergeSort
  - MergeSort - împărțim vectorul în doi subvectori care se sortează și cei doi subvectori sortați se interclasează pentru a obține vectorul sortat (interclasarea se realizează după cele două apeluri recursive). Împărțirea se face în doi subvectori de dimensiune egală
  - QuickSort – partiționăm vectorul în doi subvectori astfel încât atunci când cei doi subvectori sunt sortați tot vectorul este sortat (partiționarea se face înainte de cele două apeluri recursive). Partiționarea depinde de conținutul vectorului

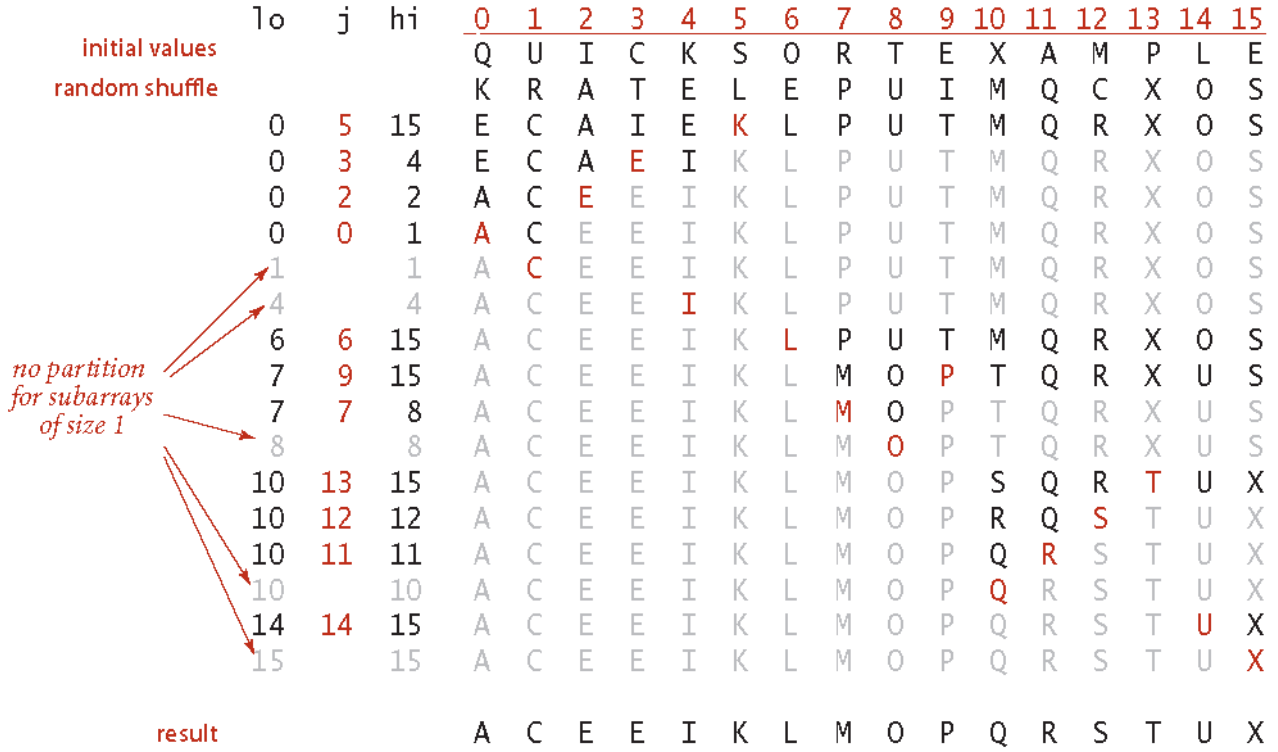
Sortare QuickSort

- QuickSort partiționare



Sortare QuickSort

- QuickSort este o funcție recursivă care sortează vectorul `a[lo..hi]` folosind funcția de partiționare care pune pe `a[j]` pe poziția finală și rearanjează restul valorilor astfel încât restul apelurilor recursive vor termina sortarea



Sortare QuickSort

```
public static void sort<T>(T[] a) where T : IComparable<T>
        {
            Util.shuffle(a); // se elimina dependenta de datele de intrare
            sort(a, 0, a.Length);
        }
private static void sort<T>(T[] a, int lo, int hi) where T : IComparable<T>
        {
            if (hi <= lo)   return;
            int j = partition(a, lo, hi);
            sort(a, lo, j - 1);
            sort(a, j + 1, hi);
        }
```

Sortare QuickSort

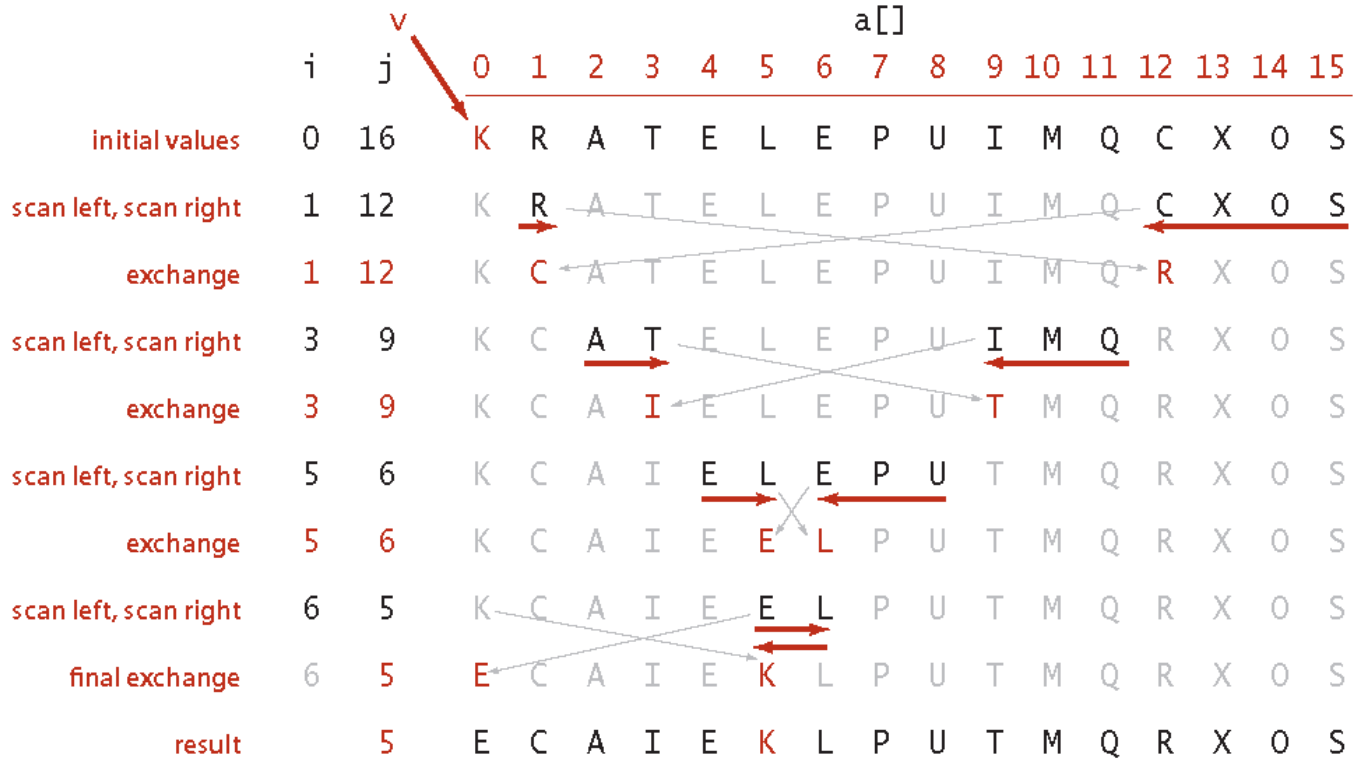
```
private static int partition<T>(T[] a, int lo, int hi) where T: IComparable<T> {
          int i = lo, j = hi + 1;
           T v = a[lo]; // elementul dupa care se face partitionarea. la sfarsitul procesului acest element va fi pe locul final in vectorul sortat
            while (true) {
                while (less(a[++i], v))   if (i == hi)  break;
                while (less(v, a[--j]))   if (j == lo)  break;
                if (i >= j)   break;
                exch(a, i, j);
            }
            exch(a, lo, j); // punem pe v in pozitia finala j
            return j;       // a[lo..j-1] <= a[j] <= a[j+1..hi]
        }
```

Sortare QuickSort

- Dificultatea algoritmului QuickSort este procesul de partiționare care rearanjează elementele vectorului a.î. următoarele 3 condiții sunt îndeplinite
  - `a[j]` este în poziția finală
  - Nici o valoarea din intervalul `a[lo..j-1]` nu este mai mare decât `a[j]`
  - Nici o valoarea din intervalul `a[j+1..hi]` nu este mai mică decât `a[j]`
- QuickSort este un algoritm randomizat pentru că înainte de sortare se face operația `shuffle()` (amestecare aleatorie a elementelor vectorului)

Sortare QuickSort

- Rulare a operației de partiționare



Sortare
QuickSort – comentarii

- Partiționare in-place – dacă am folosi un vector suplimentar operația de partiționare ar fi foarte simplă. Aici nu se folosește un vector suplimentar pentru că procedura in-place nu este foarte complicată
- Păstrarea intervalului – dacă cel mai mic sau cel mai mare element din vector este elementul de partiționare trebuie să asigurăm faptul că nu ieșim cu indicii din limitele vectorului. Am verificat în mod explicit această situație dar testul `(j == lo)` este redundant
- Operația `shuffle()` pune elementele vectorului într-o ordine aleatorie. Datorită faptului că toate elementele vectorului sunt tratate uniform cei doi subvectori sunt de asemenea în ordine aleatorie. O alternativă ar fi alegerea aleatorie a valorii după care se face partiționarea

Sortare
QuickSort – comentarii

- În implementarea QuickSort trebuie asigurat faptul că bucla `while(true)` se va încheia. Indicii i și j trebuie să se intersecteze. Trebuie ținut cont și de faptul că vectorul poate conține și alte elemente egale cu valoare de partiționare (pivotul)
- Procesul de scanare (stânga-dreapta, dreapta-stânga) îl oprim când găsim o cheie egală cu pivotul. Altfel ar rezulta un timp de rularea pătratic pentru anumite aplicații. Tratarea unui vector cu multe chei egale se poate face cu o strategie mai bună
- Trebuie asigurată terminarea procesului recursiv

Sortare
QuickSort – caracteristici de performanță

- Agoritmul QuickSort a fost analizat intens dpdv matematic de-a lungul timpului și analizele au fost validate prin experimente empirice
- Bucla interioară a QuickSort este foarte rapidă întrucât se face doar o comparație cu o valoare fixă (spre deosebire MergeSort și ShellSort efectuează și deplasări de elemente în bucla interioară). Această simplitate este cea care face ca algoritmul QuickSort să fie rapid
- QuickSort face puține comparații. Eficiența depinde de cât de bine este împărțit vectorul de operația de partiționare – ceea ce depinde de valoarea pivotului
- Poziția în care ajunge pivotul poate fi în mod echiprobabil oriunde în cadrul vectorului (pentru un vector de elemente aleatoare)

Sortare
QuickSort – caracteristici de performanță

- În cel mai bun caz pivotul ajunge la mijlocul vectorului
- În acest caz avem relația C(N) = 2C(N/2) + N cu soluția C(N) ~ N log N pentru numărul de comparații
- În medie poziția pivotului va fi la mijlocul vectorului.
- Dacă ținem probabilitate exactă a poziției pivotului relația de recurență va fi mai complicat de exprimat și de rezolvat dar rezultatul final va fi similar
- **Propoziție:** QuickSort folosește în medie ~2N log N comparații (și 1/6 din această valoarea interschimbări) pentru a sorta un vector nu N chei aleatorii distincte

Sortare
QuickSort – caracteristici de performanță

- **Demonstrație:**
  - Fie C(N) numărul mediu de comparații necesar pentru a sorta un vector de lungime N
  - C(0) = C(1) = 0
  - C(N) = N + 1 + [C(0) + C(1) + … + C(N-1)] / N + [C(N-1)+C(N-2)+…+C(0)] / N, pentru N > 1
  - C(N) = costul partiționării (N+1) + costul mediu pentru sortarea subvectorului stâng + costul mediul pentru sortarea subvectorului drept (dimensiunea celor doi subvectori în mod echiprobabil poate fi orice valoare de la 0 la N-1)
  - N C(N) = N(N+1) + 2 [C(0) + C(1) + … + C(N-1)] , pt N
  - (N-1) C(N-1) = (N-1)N + 2 [C(0) + C(1) + … + C(N-2)] , pt N-1 și făcând diferența obținem
  - N C(N) - (N-1) C(N-1) = 2N + 2C(N-1), împărțim cu N(N+1)

Sortare
QuickSort – caracteristici de performanță

- C(N)/(N+1) = C(N-1)/N + 2/(N+1)
- C(N) ~ 2(N+1)(1/3 + 1/4 + …1/(N+1)) ~ 2N ln N ≈ 1.39 N lg N
- Numărul mediu de comparații este cu 39% mai mare decât în cel mai bun caz □

- Atunci când cheile nu sunt distincte, o analiză precisă este mai greu de făcut, dar numărul mediu de comparații nu va fi mai mare decât C(N)
- QuickSort poate fi foarte ineficient dacă partițiile nu sunt echilibrate
- Din acest motiv vectorul este permutat aleatoriu înainte de a sorta. În acest fel probabilitatea de a obține partiții neechilibrate este neglijabilă în practică
- **Propoziție:** În cel mai rău caz QuickSort folosește ~N^2/2 comparații, dar permutarea aleatorie ne va proteja de această situație

Sortare
QuickSort – îmbunătățiri algoritmice

- QuickSort este mai lent decât InsertionSort pentru vectori de dimensiune mică
- Prin urmare `if (hi <= lo)` poate fi înlocuit
  `if(hi <= lo + M) { InsertionSort.sort(a, lo, hi); return; }`
- Valoarea lui `M` depinde de sistem dar orice valoarea între 5 și 15 poate da rezultate bune
- Partiționarea după valoarea medie din trei valori alese aleatoriu. Dacă a, b, c sunt trei valori alese aleatoriu din vector se va alege ca pivot valoarea din cele 3 care are proprietatea că este mai mare decât una din celelalte două și mai mică decât cealaltă din cele două: x <= pivot <= y unde x = Min(a, b, c), y = Max(a,b,c)

Sortare
QuickSort – îmbunătățiri algoritmice

- Sortarea unui vector cu multe chei egale poate fi îmbunătățită (vector mare cu puține chei distincte).
- Pentru un astfel de vector versiunea inițială a QuickSort va rula recursiv pentru mulți vectori în care toate elementele sunt egale (ineficient)
- Vom partiționa vectorul în trei părți:
  - Elementele mai mici decât pivotul
  - Elementele egale cu pivotul
  - Elementele mai mari decât pivotul

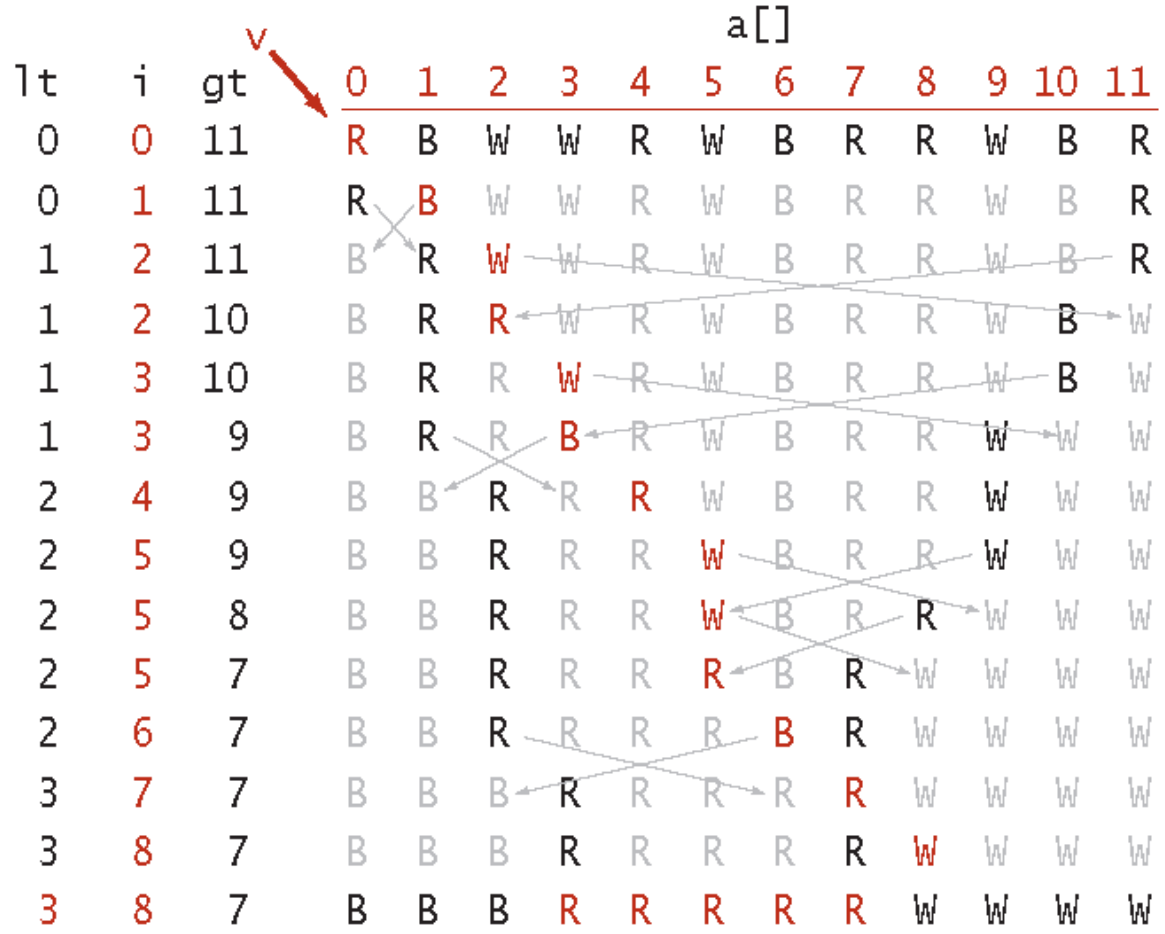
Sortare
QuickSort – îmbunătățiri algoritmice

- E.W.Dijkstra a popularizat exercițiul de partiționare în 3 sub forma problemei *Drapelului Olandez* (sortarea unui vector cu 3 elemente distincte ce corespund celor 3 culori ale drapelului) http://en.wikipedia.org/wiki/Dutch_national_flag_problem
- Problema are o soluție liniară vectorul fiind parcurs o singură dată de la stânga la dreapta
- Vom avea trei indecși `i, lt` și `gt` pentru care vom întreține următoarele relații:
  - `a[lo..lt-1]` mai mici decât v (pivotul)
  - `a[gt+1..hi]` mai mari decât v
  - `a[lt..i-1]` egale cu v
  - `a[i..gt]` nu sunt încă investigate

Sortare
QuickSort – îmbunătățiri algoritmice

```
private static void sort3Way<T>(T[] a, int lo, int hi) where T : IComparable<T>  {
            if (hi <= lo)  return;
            int lt = lo, i = lo + 1, gt = hi;
            T v = a[lo];
            while (i <= gt)  {
                int cmp = a[i].CompareTo(v);
                if (cmp < 0) exch(a, lt++, i++);
                else if (cmp > 0) exch(a, i, gt--);
                else i++;
            }
            // a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi].
            sort(a, lo, lt - 1);
            sort(a, gt + 1, hi);
        }
```

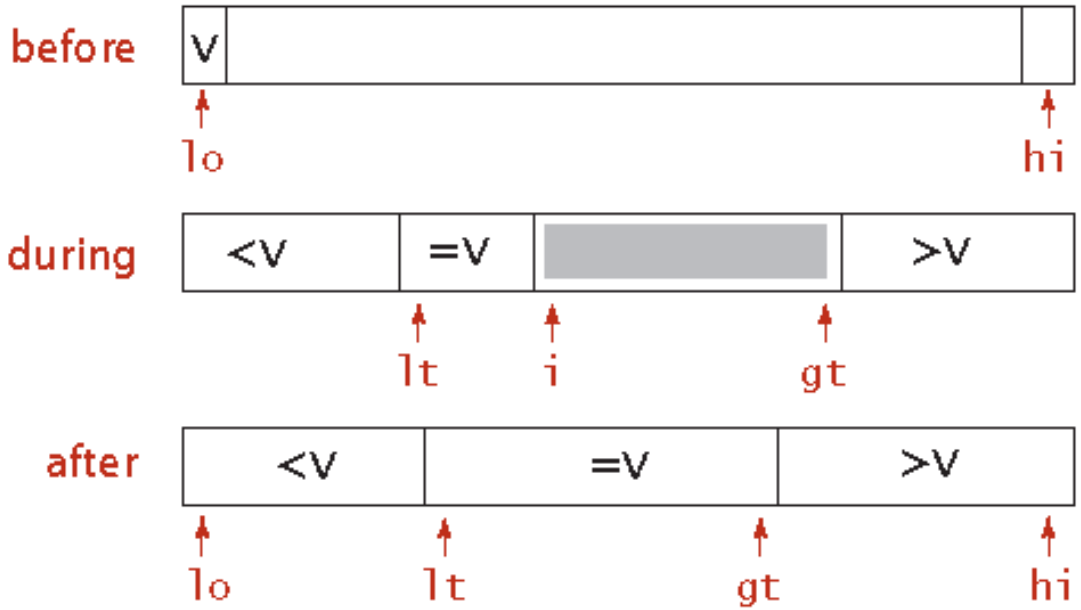
Sortare
QuickSort – îmbunătățiri algoritmice



Trace pentru partiționarea în 3

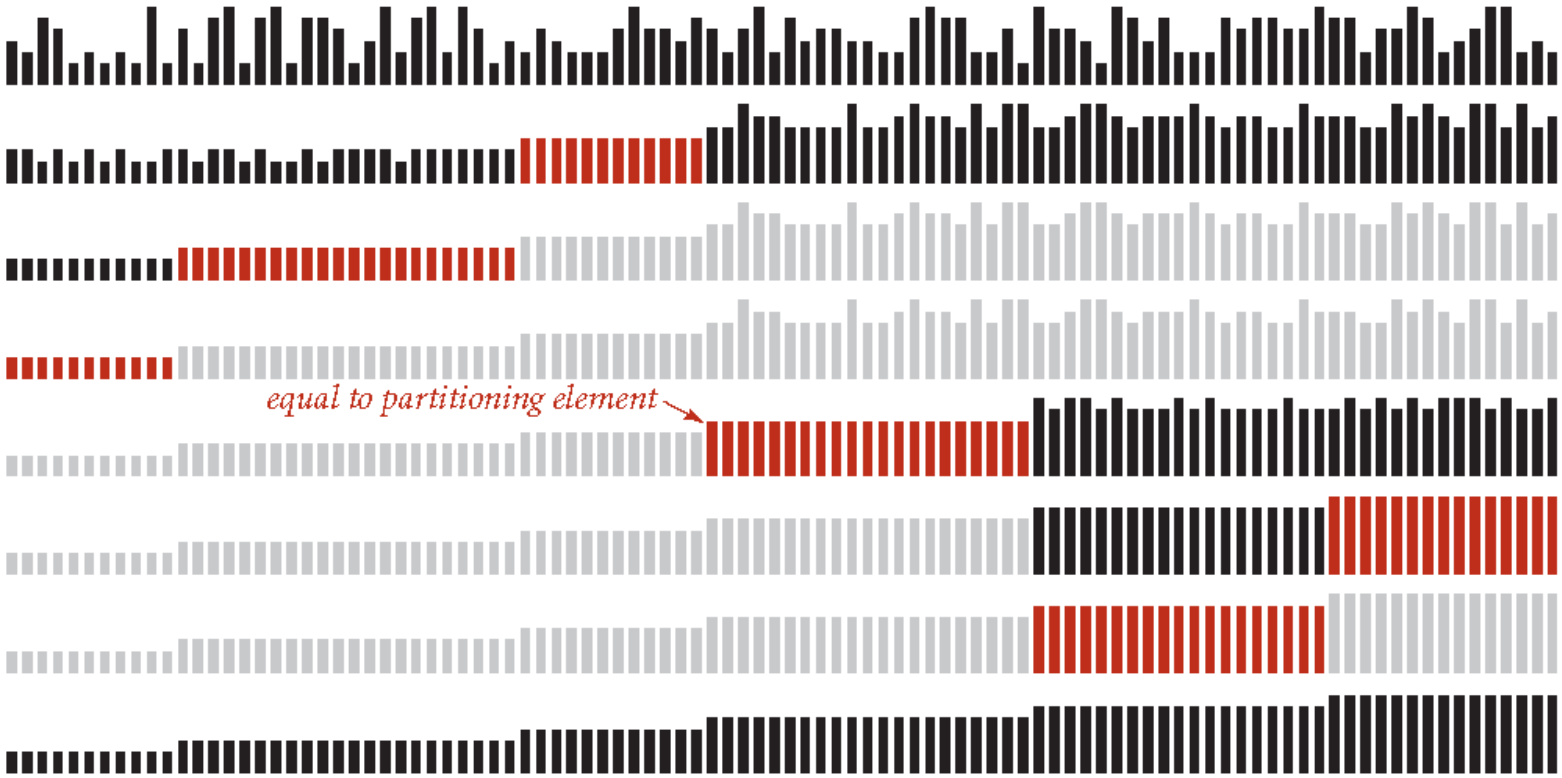
Sortare
QuickSort – îmbunătățiri algoritmice

- Partiționare în 3
- Valoarea `gt-i` va scădea la fiecare iterație prin urmare bucla se încheie
- Numărul de interschimbări este mult mai mare față de partiționarea în 2 atunci când avem puține chei duplicate



Sortare
QuickSort – îmbunătățiri algoritmice

- QuickSort cu partiționare în 3 pentru un vector cu puține chei distincte



Sortare QuickSort

- Pentru un vector cu un număr constant de valori distincte distribuite aleatoriu
  - MergeSort este linearitmic
  - QuickSort3Way este liniar
- O versiune îmbunătățită de QuickSort va rula mai eficient pe orice sistem în comparație cu orice alt algoritm de sortare bazat pe comparații
- Este algoritmul folosit pe scară largă în ziua de astăzi pe toate infrastructurile de calcul
- Atât modelele matematice cât și experimentele realizate au demonstrat superioritatea QuickSort

Coada cu prioritate

- Coada cu prioritate = structură de date care are două operații fundamentale
  - Eliminarea maximului
  - Inserarea unui nou element
- Similar cu coada (eliminarea celui mai vechi element) și cu stiva (eliminarea celui mai nou element)
- Cele două operații trebuie implementate **eficient**
- Implementarea naivă necesită timp liniar pentru una sau ambele operații
- Implementare eficientă bazată pe *binary heap*; elementele se păstrează într-un vector special care permite realizarea celor două operații în timp logaritmic

Coada cu prioritate

- Aplicații ale cozilor cu prioritate
  - Gestionarea mai multor aplicații ce rulează simultan pe un sistem de operare și care au nevoie de timp de procesor – multitasking – unele procese au prioritate mai mare decât altele
  - Sisteme de simulare – unde cheile sunt mărci de timp pentru momentul în care trebuie să aibă loc un eveniment, iar evenimentele trebuie procesate în ordine cronologică
  - Sortare – inserăm elemente în coada cu prioritate (MinHeap) și eliminăm succesiv cel mai mic element obținând lista elementelor în ordine crescătoare – *HeapSort*
  - În teoria grafurilor facilitează implementarea unor algoritmi de căutare
  - Algoritmi pentru compresia datelor

Coada cu prioritate

- Coada cu prioritate = tip abstract de date; reprezintă o mulțime de valori și operațiile pe acele valori
- Vom defini operațiile asupra cozii cu prioritate prin specificarea unui API
- Operațiile principale asupra cozii sunt eliminarea maximului și inserarea unui nou element
- Compararea cheilor se face cu metoda `less()` – la fel ca și la algoritmii de sortare
- Vom defini constructori și alte metode ajutătoare
- Vom avea o implementare generică cu un parametru de tip `Key` ce implementează interfața `IComparable<Key>`

Coada cu prioritate

- API pentru coada cu prioritate

```
class MaxPQ<Key> where Key: IComparable<Key>
    public MaxPQ(int initCapacity)
    public MaxPQ(): this(1)
    public MaxPQ(Key[] keys)
    public MaxPQ(int initCapacity, IComparer<Key> comparator)
    public Key max() // Intoarce cea mai mare cheie din coada cu prioritate
    public Key delMax() // Elimina din coada cu prioritate cel mai mare element si il intoarce ca rezultat
    public void insert(Key x) // Adaugam o noua cheie in coada cu prioritate
    public bool isEmpty() // Testeaza daca e goala coada cu prioritate
    public int size() // Intoarce numarul de chei din coada cu prioritate
```

Coada cu prioritate

- API pentru coada cu prioritate

```
class MinPQ<Key> where Key: IComparable<Key>
    public MinPQ(int initCapacity)
    public MinPQ(): this(1)
    public MinPQ(Key[] keys)
    public MinPQ(int initCapacity, IComparer<Key> comparator)
    public Key min() // Intoarce cea mai mica cheie din coada cu prioritate
    public Key delMin() // Elimina din coada cu prioritate cel mai mic element si il intoarce ca rezultat
    public void insert(Key x) // Adaugam o noua cheie in coada cu prioritate
    public bool isEmpty() // Testeaza daca e goala coada cu prioritate
    public int size() // Intoarce numarul de chei din coada cu prioritate
```

Coada cu prioritate

- **Aplicație:** se dă o listă de N valori (N este foarte mare – ar putea fi considerat nelimitat) și se cere să se determine cele mai mici M valori din cele N
- **Rezolvare:** pentru a putea rezolva eficient această problemă folosim un MaxPQ în care păstrăm cele mai mici M elemente din cele investigate până la un moment. Dacă o valoarea este mai mică decât elementul maxim din MaxPQ (și dacă MaxPQ conține deja M elemente) atunci elementul maxim este eliminat și se introduce noua valoare
- Obs. 1. Sortarea celor N elemente nu e posibilă întrucât N este foarte mare (elementele nu pot fi stocate)
- Obs. 2. Compararea fiecărui element cu cele mai mici M elemente este fezabilă (dpdv computațional) doar dacă M este mic

Coada cu prioritate

```
public static void Main(string[] args)       {
            int M = 5; // cele mai mari 5 elemente din fisier
            MinPQ<int> pq = new MinPQ<int>(M + 1);
            StreamReader fs = new StreamReader("largeT.txt");
            string line;
            while (!fs.EndOfStream)  {
                line = fs.ReadLine();
                pq.insert(int.Parse(line));
                // eliminam valoarea minima din coada cu prioritate daca sunt M+1 elemente in coada
                if (pq.size() > M)   pq.delMin();
            } // cele mai mari M elemente sunt in coada
            Stack<int> stack = new Stack<int>(); // afisam elementele din CP in ordine inversa
            foreach (var item in pq)         stack.push(item);
            foreach (var item in stack)      Console.WriteLine(item);
    }
```

Coada cu prioritate

- Costul găsirii celor mai mari/mici M elemente dintr-o listă de N elemente

Client	Ordinul de creștere	
	Timp	Spațiu
Client care sortează	N * log N	N
Client MinPQ care folosește implementare elementară	N * M	M
Client MinPQ care folosește o implementare bazată pe heap	N * log M	M

Coada cu prioritate

- Implementări elementare pentru CP
  - Vector nesortat (abordare *lazy*): operația insert() inserează elementul nou pe prima poziție liberă din vector (constant - eficient); operația delMax() interschimbă cel mai mare element din vector cu elementul de pe ultima poziție și îl elimină (liniar - ineficient) [lazy – efectuăm operația de căutare a maximului atunci când e nevoie]
  - Vector sortat (abordare *eager*): operația insert() mută elementele mai mari decât elementul ce este inserat o poziție la dreapta pentru a-i face loc acestui element pentru a fi inserat la poziția corectă a.î. vectorul este în ordine crescătoare (liniar - ineficient); operația delMax() se poate face eficient pentru că elementul maxim este pe ultima poziție (constant - eficient); [eager – operația de determinare a maximului se face înainte de a avea nevoie de maxim]
- Pentru aceste două implementări putem folosi cod de redimensionare a vectorului astfel încât să asigurăm faptul că cel puțin un sfert din vector este întotdeauna ocupat și nu apare niciodată situația de depășire a limitelor

Coada cu prioritate

Ordinul de creștere în cel mai rău caz pentru diverse implementări pentru coada cu prioritate

Structura de date	Insert	Eliminare maxim
Vector sortat	N	1
Vector nesortat	1	N
Heap	log N	log N
Imposibil	1	1

Coada cu prioritate

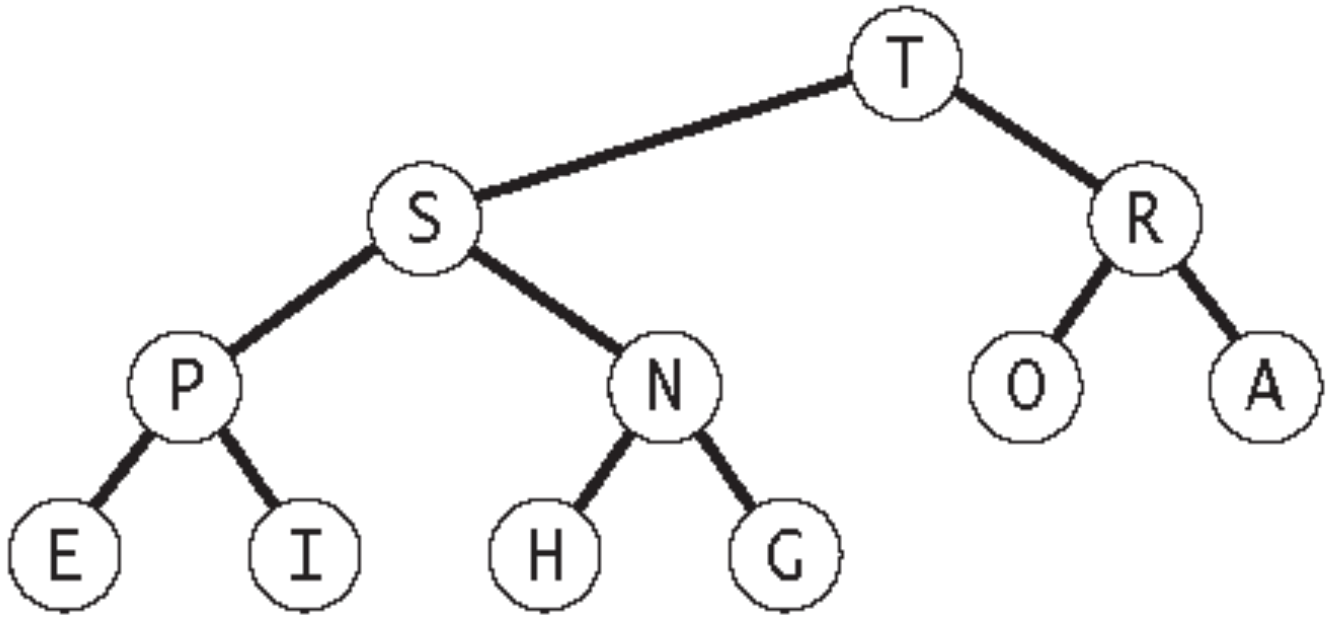
- **Heap binar** – structură de date care permite realizarea eficientă a operațiilor elementare pentru coada cu prioritate
- Într-un heap se garantează că fiecare cheie este mai mare decât alte două chei de la o anumită poziție bine precizată. Fiecare din cele două chei sunt mai mari (sau egale) decât alte două chei de la anumite poziții etc.
- Această ordine se poate vizualiză ușor dacă privim heap-ul ca un arbore binar
- **Definiție:** un heap binar este un arbore binar în care cheia din fiecare nod este mai mare sau egală decât cheia din nodurile copil. (echivalent: cheia din orice nod este mai mică sau egală decât cheia din nodul părinte)
- **Propoziție:** cea mai mare cheie dintr-un heap se găsește în rădăcina arborelui

Coada cu prioritate

- Reprezentări pentru heap:
  - Într-o structură de date dinamică cu noduri și legături ar trebui să avem în fiecare nod 3 legături – una la nodul părinte și două la nodurile copil
  - Un arbore binar complet = arbore în care se completează pe niveluri de la stânga la dreapta. Pot fi reprezentați cu ajutorul unui vector fără a mai fi nevoie de a întreține legături complicate.
  - În vector rădăcina arborelui binar complet va fi pe poziția 1, cei doi copii ai rădăcinii pe pozițiile 2, 3, nodurile copil ale acestora pe pozițiile 4, 5 respective 6, 7 etc.

Coada cu prioritate

- Un arbore binar complet ordonat heap

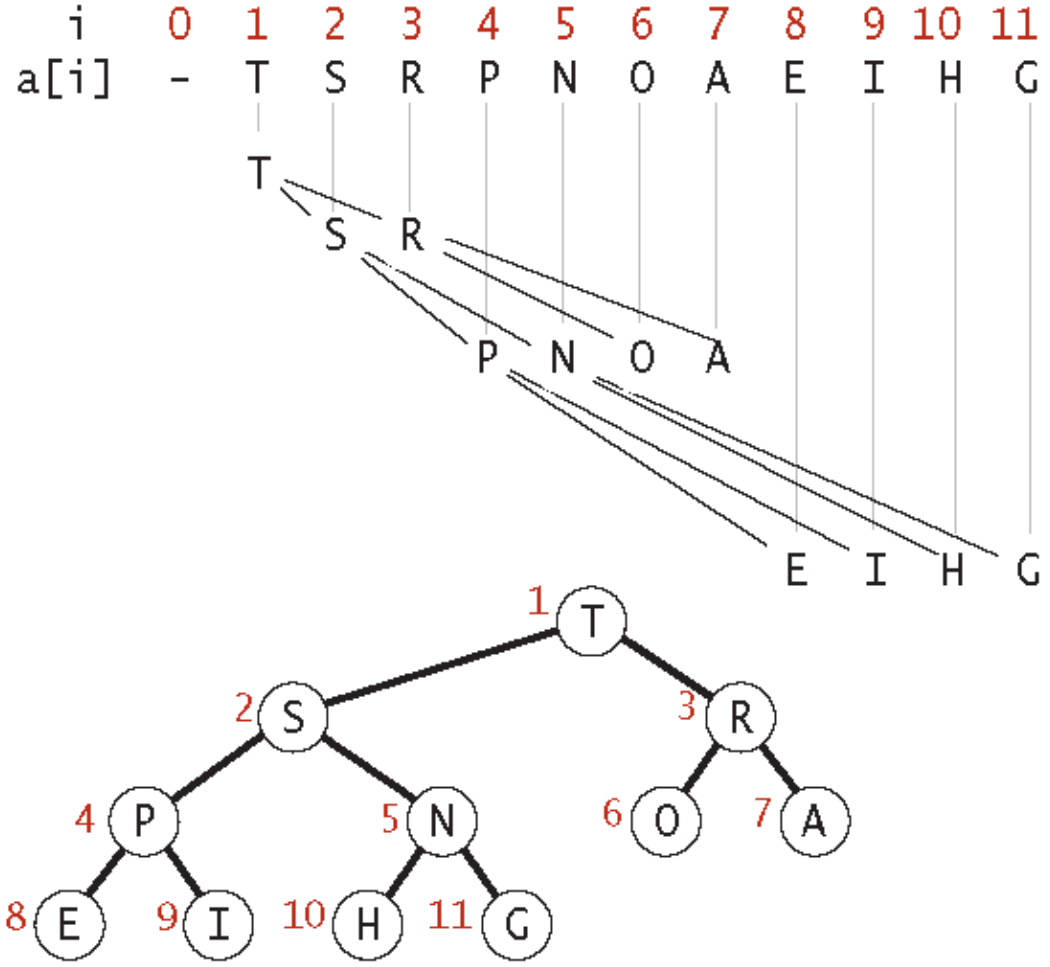


Coada cu prioritate

- **Definiție:** Un heap binar este o colecție de chei aranjate într-un arbore binar complet ordonat heap, reprezentate într-un vector ordonat după nivelurile arborelui (fără să folosim indexul 0 din vector)
- Într-un heap părintele nodului de pe poziția k se află pe poziția k / 2
- Într-un heap copii unui nod de pe poziția k se află pe pozițiile 2*k respectiv 2*k + 1
- Arborii binari compleți reprezentați ca și vectori ne permit să implementăm eficient operațiile principale ale cozii cu prioritate
- Operațiile de inserare și eliminare a elementului maxim se vor realiza în timp logaritmic

Coada cu prioritate

- Reprezentarea unui heap într-un vector
- Înălțimea unui arbore binar complet de dimensiune N este log2 N



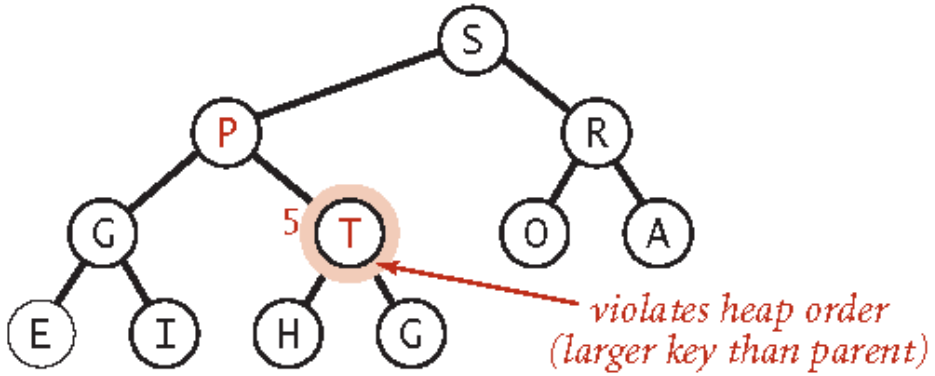
Coada cu prioritate

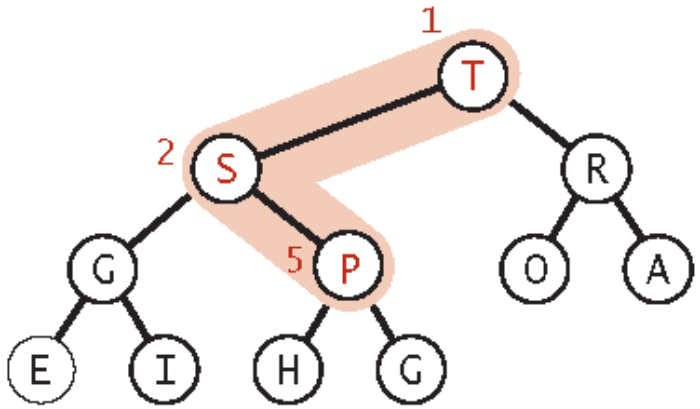
- Algoritmi asupra heap
- Sunt anumite situații când se modifică o cheie (sau se adaugă o nouă cheie la sfârșitul vectorului) ceea ce duce la violarea condiției de heap (cheia din fiecare nod este mai mică decât cheia din nodul părinte)
- Într-o astfel de situație trebuie să aplicăm o operație numit *reheapify* care restabilește ordinea în heap.
- Sunt două situații:
  - Valoarea unei chei se mărește (sau se adaugă o nouă cheie la sfârșit) – va trebui să migrăm cheia în sus în arbore cu o operație numită *swim*
  - Valoarea unei chei scade – va trebui să migrăm cheia în jos pentru a restabili ordinea în heap cu o operație numită *sink*

Coada cu prioritate

- Bottom-up reheapify (swim)

```
private void swim(int k)
{
    while (k > 1 && less(k / 2, k))
    {
        exch(k, k / 2);
        k = k / 2;
    }
}
```

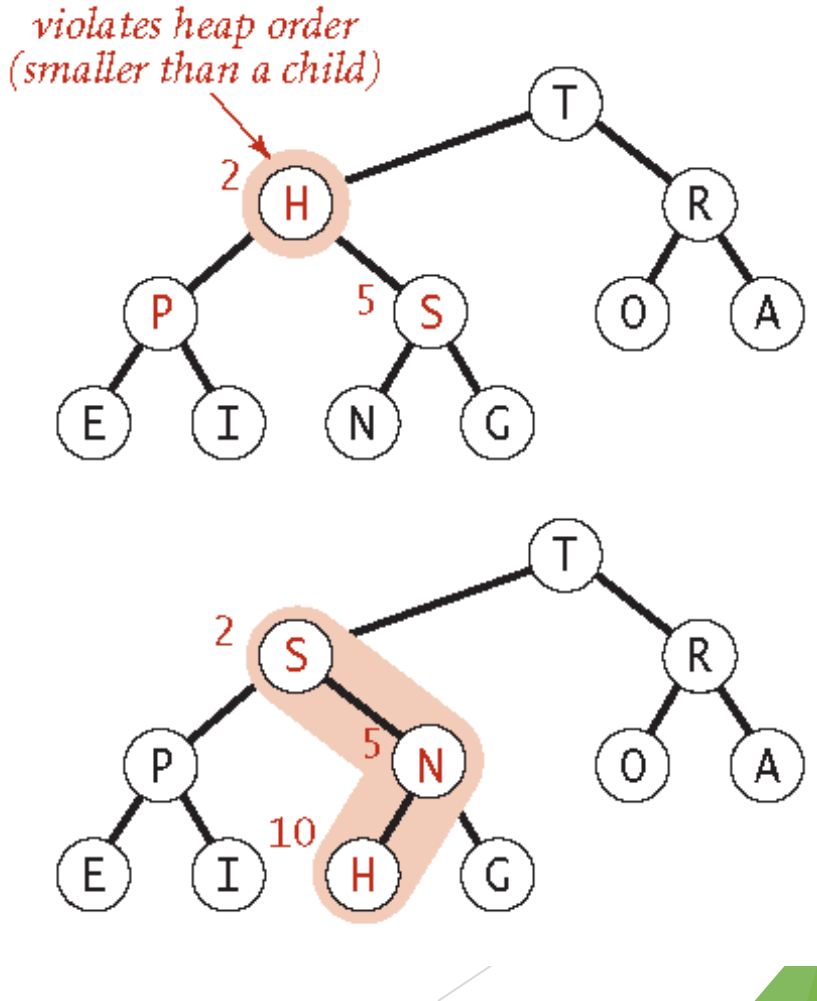




Coada cu prioritate

- Top-down reheapify (sink)

```
private void sink(int k){
    while (2 * k <= N){
        int j = 2 * k;
        if (j < N && less(j, j + 1))
            j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

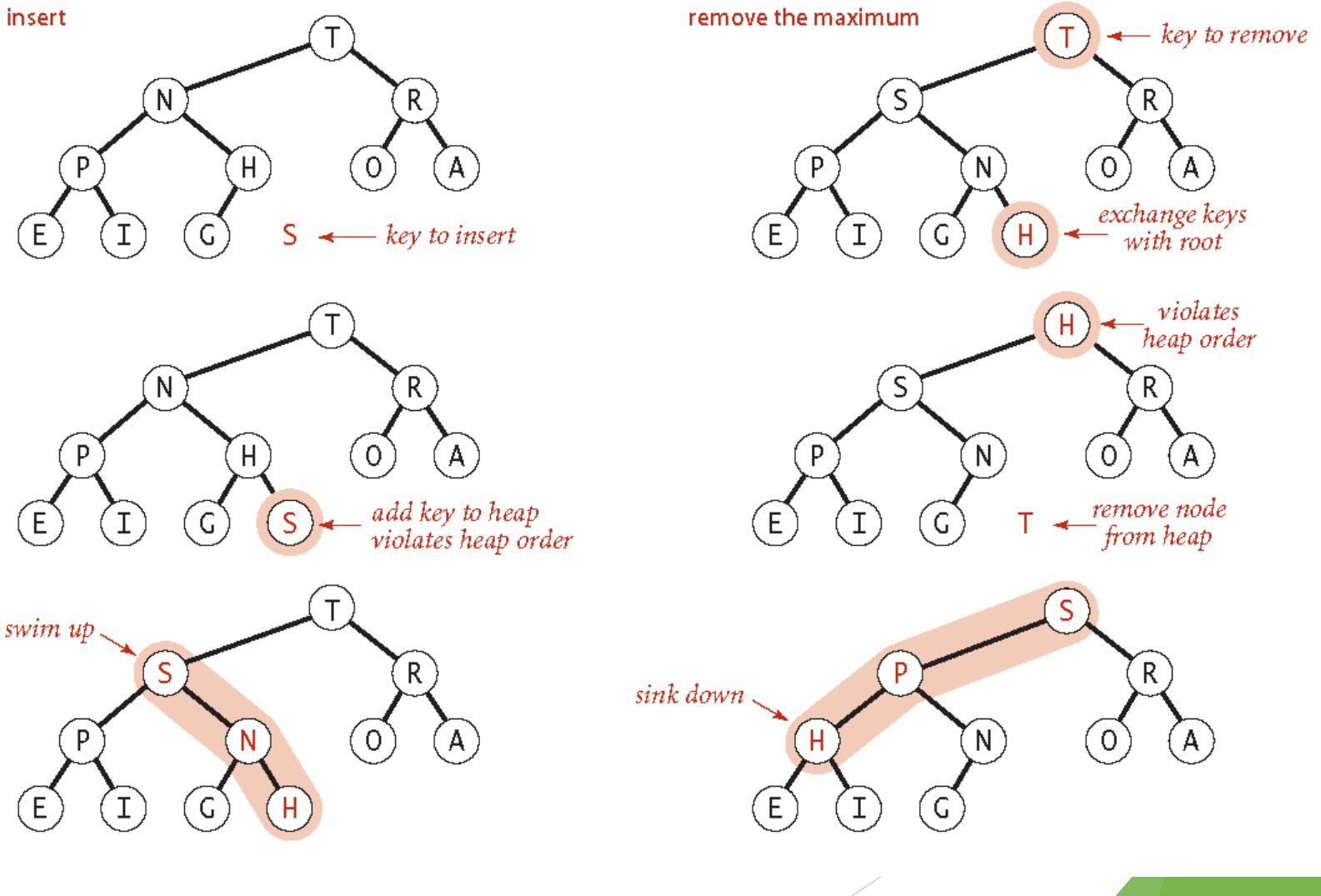


Coada cu prioritate

- Operațiile `sink()` și `swim()` permit implementarea eficientă a API-ului pentru coada cu prioritate
  - Inserarea unei noi valori – se face la sfârșit, se mărește dimensiunea heap-ului cu 1 și noii valori se aplică operația `swim()` pentru a restabili proprietatea de heap
  - Eliminarea maximului – maximul este pe poziția 1; în locul lui se pune elementul de pe ultima poziție și se aplică asupra acestuia operația `sink()`
- Ambele operații se vor face în timp logaritmic (eficient)
- MaxPQ.cs, MinPQ.cs, MaxMinPQClient.cs, TopM.cs

Coada cu prioritate

- Operații asupra heap



Coada cu prioritate

- **Propoziție:** Într-o coadă cu prioritate cu N chei, algoritmul de inserare în heap necesită cel mult 1 + log2 N operații de comparare iar algoritmul de eliminarea a maximului necesită cel mult 2 log2 N operații de compare.
- Pentru aplicații în care avem un număr foarte mare de operații *inserare*, *eliminare maxim*, intercalate, implementarea pe bază de heap a cozii cu prioritate permite realizarea ambelor operații în timp logaritmic ceea ce duce la rezolvarea problemei spre deosebire de soluțiile liniare care nu permit rezolvarea problemei atunci când numărul de elemente procesate este foarte mare

Coada cu prioritate

- Codul poate fi modificata cu ușurință pentru a crea heap ternar (sau n-ar)
- Într-un arbore ternar complet fiecare nod are trei descendenți direcți
  - Un element de la poziția k va fi mai mare sau egal decât elementele de la pozițiile 3k-1, 3k și 3k+1 (vectorul este indexat de la 1).
  - Părintele elementului de la poziția k va fi la poziția (k+1)/3
- În acest caz înălțimea arborelui va fi log3 N (N numărul de elemente din heap); înălțimea va fi mai mică
- Costul pentru găsirea celui mai mic descendent va fi mai mare;
- Se face un compromis în funcție de frecvența operațiilor care ne așteptăm să fie utilizate.

Coada cu prioritate indexată
TODO

- Aplicație: Multiway merge (interclasare a N stream-uri)

HeapSort

- Inventat de J.W.J Williams și rafinat de R.W. Floyd în 1964
- Putem folosi un heap pentru a implementa un algoritm de sortare
- HeapSort are două faze
  - Construcția heap-ului – se reorganizează vectorul original ca un heap
  - SortDown – eliminăm elementele din heap în ordine descrescătoare pentru a crea vectorul sortat
- Pentru ordonarea crescătoare vom vom folosi un MaxPQ
- Sortarea se va face in-place; heap-ul va fi creat chiar în vectorul care se sortează
- Crearea unui heap dintr-un vector de elemente aleatorii se poate face parcurgând-ul de la stânga la dreapta și aplicând operația swim()

HeapSort

- Crearea unui heap dintr-un vector de elemente aleatorii se poate face parcurgând-ul de la stânga la dreapta și aplicând operația swim().
- Această operație se poate realiza în timp N log N.
- O altă metodă este de a parcurge vectorul de la dreapta la stânga și aplicarea metodei sink() pentru a crea subheap-uri (dacă cei doi subarbori ai unui nod părinte sunt heap atunci apelul metodei sink() pe acel nod va face din arborele cu rădăcina în nodul părinte un heap)
- Cu această metodă putem începe de la mijlocul vectorului înspre început pentru că toate elementele din dreapta mijlocului sunt frunze în arbore (heap de dimensiune 1)

HeapSort

- **Propoziție:** construcția unui heap prin apeluri la sink() folosește mai puțin de 2N operații de comparație și mai puțin de N interschimbări pentru a crea un heap nu N elemente

```
public static void sort<T>(T[] pq) where T: IComparable<T>
        {
            int N = pq.Length;
            for (int k = N / 2; k >= 1; k--)
                sink(pq, k, N);
            while (N > 1)
            {
                exch(pq, 1, N--);
                sink(pq, 1, N);
            }
        }
```

HeapSort

- În metodele exch() și less() indicii sunt decrementați pentru a sorta un vector cu indici de la 0 la N-1

```
private static bool less<T>(T[] pq, int i, int j) where T:IComparable<T>
        {
            return pq[i - 1].CompareTo(pq[j - 1]) < 0;
        }

private static void exch<T>(T[] pq, int i, int j) where T: IComparable<T>
        {
            T swap = pq[i - 1];
            pq[i - 1] = pq[j - 1];
            pq[j - 1] = swap;
        }
```

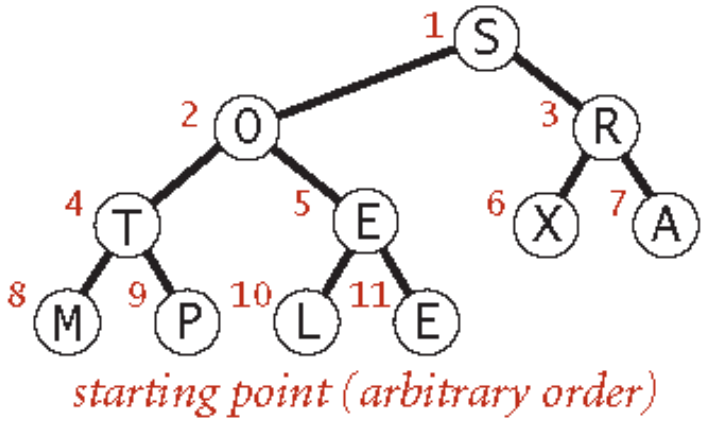
HeapSort

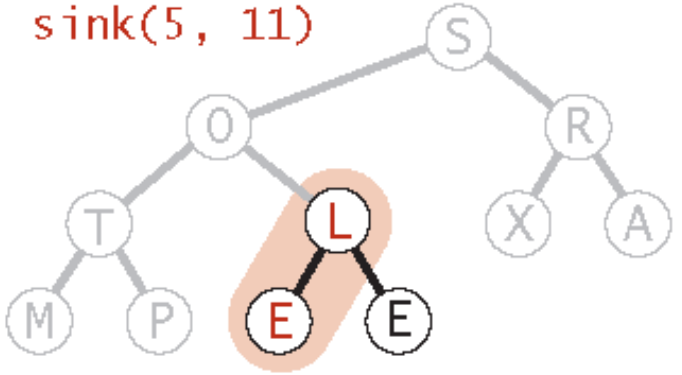
- Un trace al HeapSort
- Conținutul vectorului după fiecare operație `sink()`

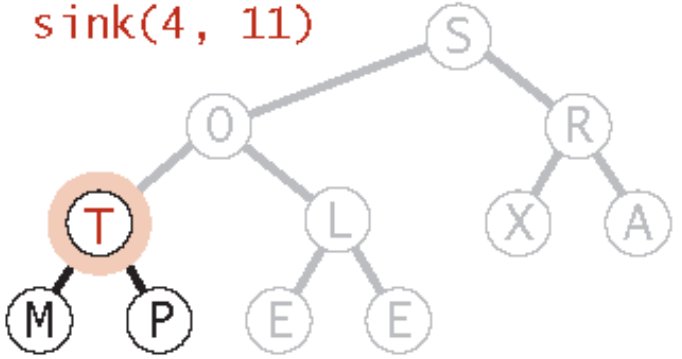
		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
sorted result			A	E	E	L	M	O	P	R	S	T	X

HeapSort

- Construcția heap

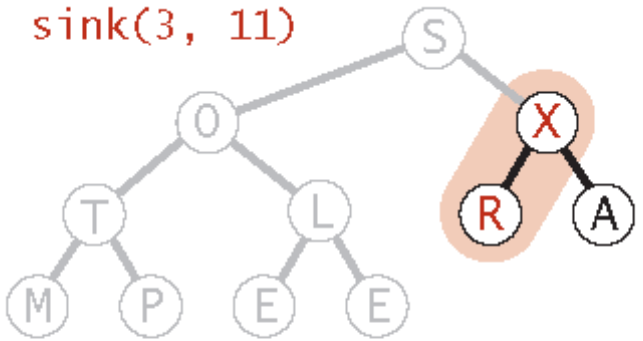


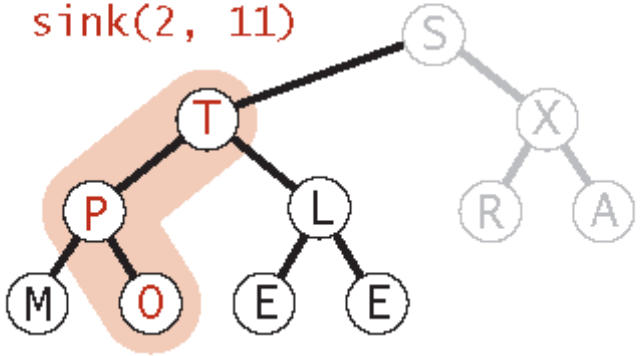


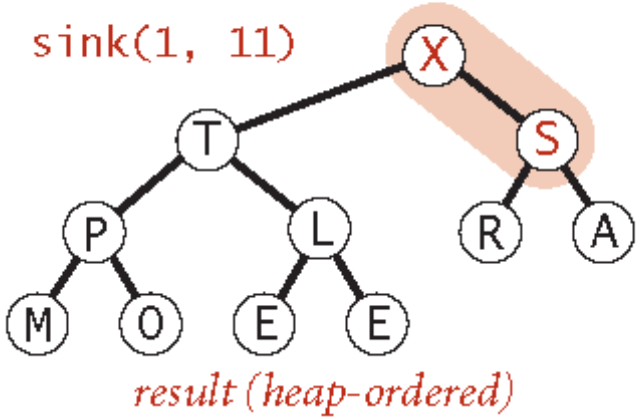


HeapSort

- Construcția heap

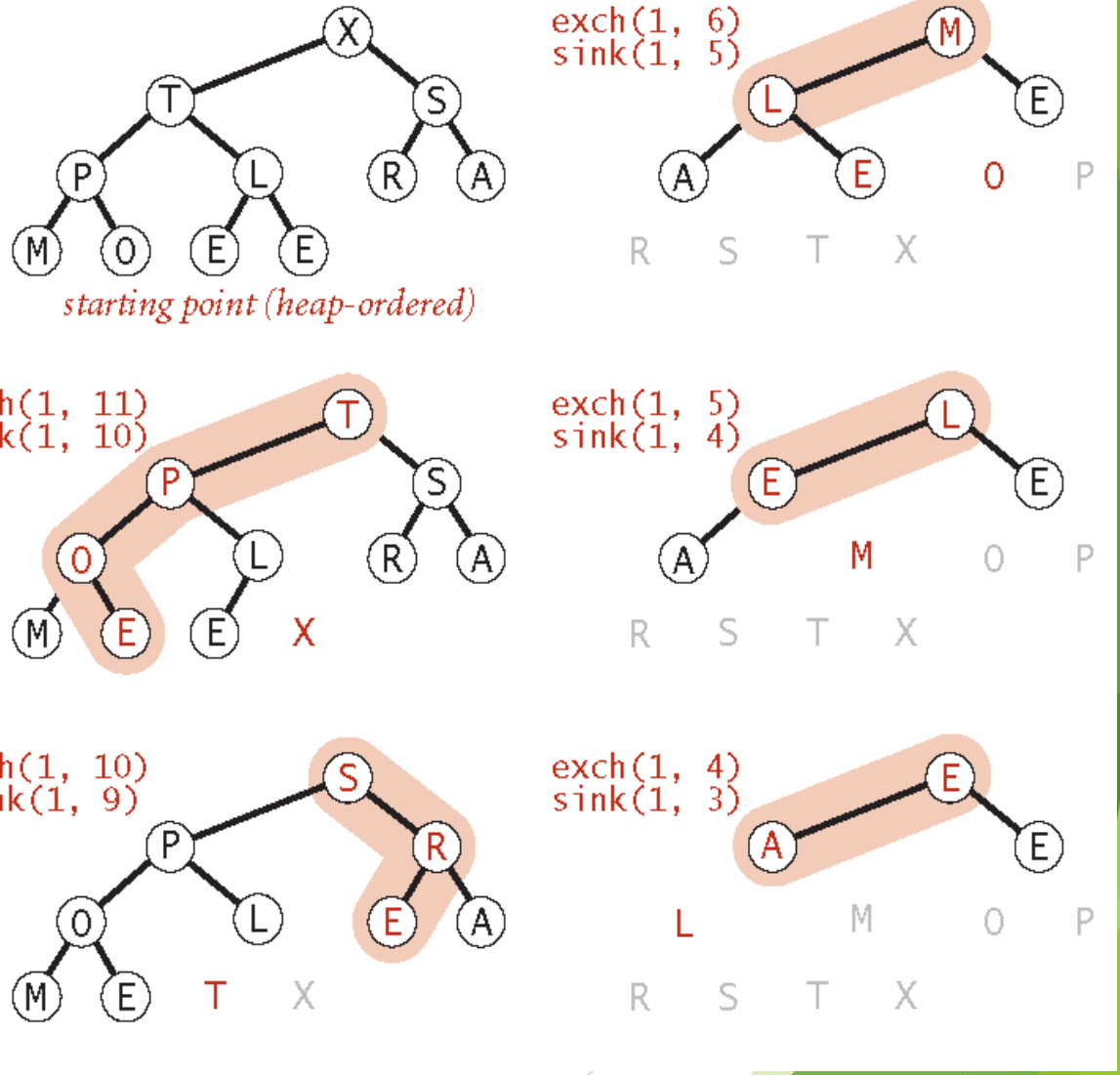






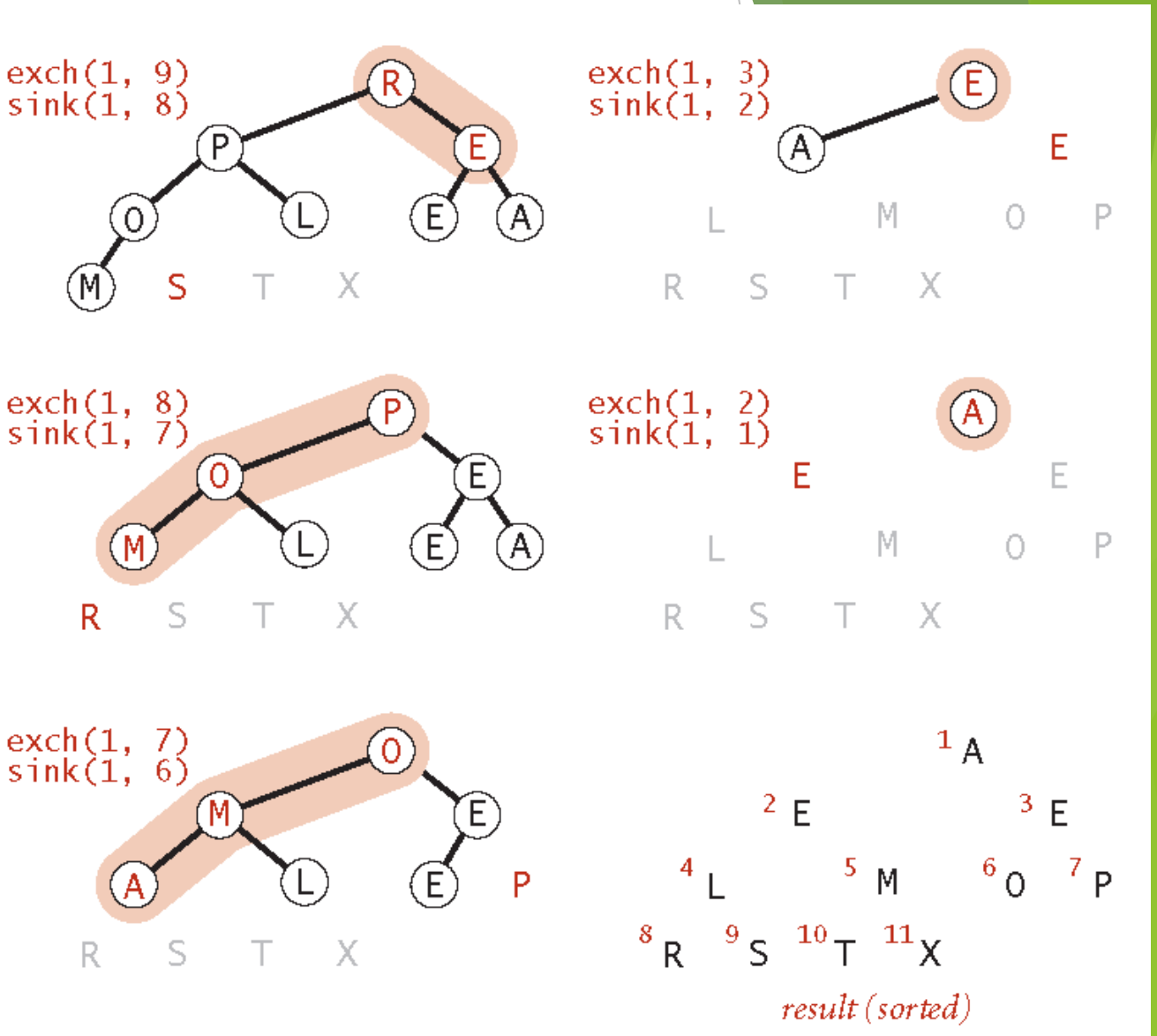
HeapSort

- Sortarea heap
- Cel mai mare element din heap este eliminat prin mutarea înspre sfârșitul vectorului
- Dimensiunea heap-ului scade cu 1
- Proprietatea heap este restabilita



HeapSort

- Sortarea heap
- **Propoziție:** HeapSort folosește mai puțin de 2N log 2N + 2N comparații (și jumătate din această valoarea interschimbări) pentru a sorta un vector de N elemente



Algoritmi de sortare – aplicații

- Un motiv pentru care sortarea este utilă = mult mai ușor să se caute un element într-un vector sortat
- Aplicații
  - agende, organizarea unor fișiere, motoare de căutare (sortare în ordinea relevanței), foi de calcul,
  - Indexul unei cărți, eliminarea elementelor care se repetă într-o listă, calcule statistice
  - Compresia datelor, grafică pe calculator, biologie computațională, optimizare combinatorială
- Algoritmii de sortare au un rol fundamental în multe alte categorii de algoritmi fiind un prim pas în organizarea datelor

Algoritmi de sortare – aplicații

- Sortarea unor elemente de tipuri diferite
- Algoritmii de sortare au fost implementați în mod generic
- Tipurile de date trebuie să implementeze interfața `IComparable<T>` care stabilește o ordine naturală pe valorile tipului de date
- Pentru a sorta și după alte criterii vom crea implementări ale interfeței `IComparer<T>` și vom folosi un comparator
- Când sortăm ceea ce se modifică sunt referințe la date. Datele propriu-zise (obiectele) rămân în memorie la aceleași adrese (*pointer sorting*)
- Cheile de sortare trebuie să fie imutabile pentru a asigura că ordinea rămâne neschimbată dacă un client ar încerca să modifice valorile cheilor

Algoritmi de sortare – aplicații

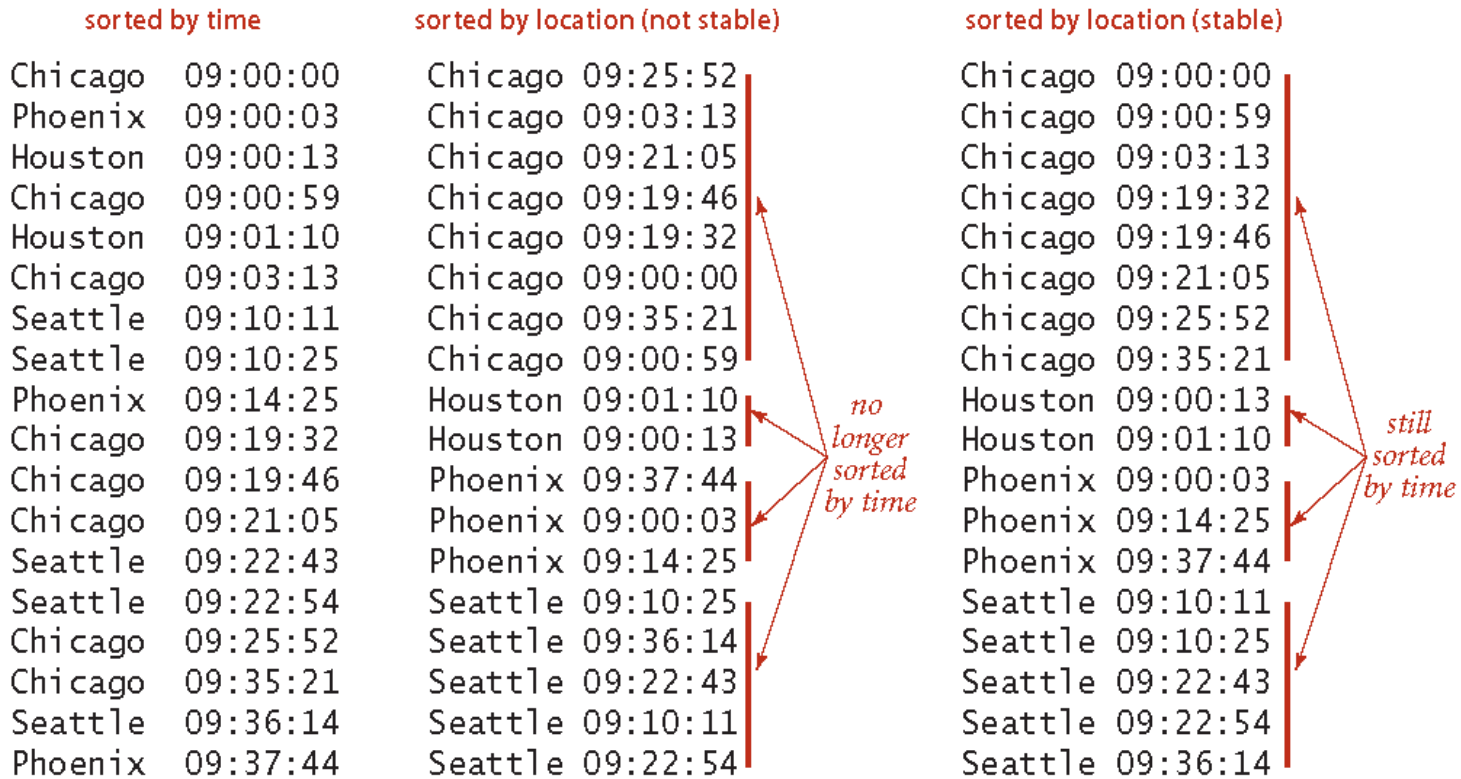
- Folosind referințe la date, interschimbările de elemente nu sunt operații costisitoare
- De asemenea operația de comparație de regulă nu este costisitoare pentru că pentru a stabili ordinea a două chei se folosește doar o mică parte din datele conținute de cheia respectivă
- Există multe situații în care dorim să folosim mai multe criterii de sortare
- Interfața `IComparer<T>` ne permite să realizăm acest lucru
- Ex. https://msdn.microsoft.com/en-us/library/bzw8611x(v=vs.110).aspx
- Transaction.cs

Algoritmi de sortare – aplicații

- **Stabilitate** – proprietate a algoritmilor de sortare care păstrează ordinea relativă a cheilor egale
- Este o proprietate utilă în multe situații
- Exemplu: o listă de obiecte (ce conține informații legate de timp și locație GPS) ordonate după timp se ordonează ulterior după locație. Dacă sortarea nu este stabilă timpii vor fi într-o ordine aleatorie.
- Algoritmi de sortare stabilă: Insertion Sort, Merge Sort
- Algoritmi de sortare instabilă: Selection Sort, Quick Sort, Shell Sort, Heap Sort

Algoritmi de sortare – aplicații

- Stabilitate când se sortează după o a doua cheie



Algoritmi de sortare – aplicații

- Care algoritm de sortare ar trebui folosit?
- Răspunsul depinde în mare măsură de detaliile aplicației și de implementare

Algoritm	Stabil ?	In place ?	Ordinul de creștere pentru a sorta N valori Timpul	spațiu suplimentar	Obs.
Selection	Nu	Da	N^2	1	
Insertion	Da	Da	Între N și N^2	1	Depinde de ordinea elementelor
ShellSort	Nu	Da	N log N? N^ (6/5)	1	
QuickSort	Nu	Da	N log N	log N	probabilistic
3-way Quick	Nu	Da	N – N log N	log N	Probabilistic, depinde de distribuția cheilor
MergeSort	Da	Nu	N log N	N	
HeapSort	Nu	Da	N log N	1	

Algoritmi de sortare – aplicații

- **Proprietate:** QuickSort este cel mai bun algoritm de sortare de uz general
- Justificare:
  - S-au făcut multe experimente pe diverse sisteme care au dus la această concluzie.
  - Bulca internă are puține instrucțiuni
  - Datele sunt accesate secvențial în vector prin urmare poate să profite de memoria cache
  - Timpul de rulare este ~c N log N unde c este o constantă mai mică decât la ceilalți algoritmi linearitmici
  - 3 Way QuickSort devine liniar pentru anumite distribuții ale cheilor care pot să apară în practică
- Dacă e importantă stabilitatea MergeSort e o alegere bunăs

Algoritmi de sortare – aplicații

- Sortarea tipurilor primitive: `int, float, double`
- Pentru tipurile primitive este indicat să redefinim rutinele de sortare fără a folosi vectori de `IComparable<T>`
- Vom sorta direct vectori de tip `int[]` sau `double[]`
- Astfel evităm stocarea referințelor și costul suplimentar al accesului la date prin referințe
- De asemenea se evită costul apelului la metodele `less()` și `CompareTo()`
- Vom folosi direct cod de forma `a[i] < a[j]`

Algoritmi de sortare – aplicații

- **Reducții:** o tehnică de bază în proiectarea algoritmilor în care folosim soluția la o problemă B pentru a rezolva problema A. (Spunem că reducem problema A la problema B)
- Sortarea este un exemplu clasic de reducție
- Soluția multor probleme (care au o soluție brută de tip ~N^2) se poate reduce la sortare.
- Se sortează prima dată datele (în timp ~N log N) iar mai apoi soluția se obține rapid de regulă printr-un algoritm liniar

Algoritmi de sortare – aplicații

- Exemple: (duplicate)
  - Există chei care se repetă într-un vector?
  - Câte chei distincte există într-un vector?
  - Care valoare apare cel mai frecvent într-un vector?
- Toate aceste probleme au o soluție pătratică imediată pentru vectori de dimensiune mică (fiecare elemente al vectorului se compară cu celelalte elemente)
- Pentru vectori de dimensiune mare algoritmul pătratic nu se poate aplica (e nefezabil computațional)
- Prima dată sortăm vectorul în timp linearitmic după care parcurgem vectorul o singur dată pentru a găsi răspunsul la întrebările de mai sus

Algoritmi de sortare – aplicații

- Numărarea cheilor distincte

```
QuickSort.sort(a);
int count = 1; // presupunem a.Length > 0.
for (int i = 1; i < a.Length; i++)
    if (a[i].CompareTo(a[i - 1]) != 0)
        count++;
```

Algoritmi de sortare – aplicații

- Se dau două permutări aleatorii ale numerelor de la 0 la N-1
- **Distanța *Kendall tau*** dintre cele două permutări = numărul de perechi de numere pentru care ordinea din cele două permutări este diferită
- Exemplu: P1 = {0, 3, 1, 6, 2, 5, 4} și P2 = {1, 0, 3, 6, 4, 2, 5} Distanța Kendall tau este 4 pentru ca perechile care nu sunt în aceeași ordine sunt (0, 1), (3, 1), (2, 4), (5, 4)
- Această valoare este folosită în multe domenii: sociologie, teoria votului, biologie moleculară, motoare de căutare pe web
- Se cere să se scrie un program eficient care să calculeze această valoare pentru două permutări

Algoritmi de sortare – aplicații

- **Găsirea valorii medii** dintr-un vector = valoarea x pentru care jumătate din elementele vectorului sunt mai mici decât x și jumătate sunt mai mari decât x
- Este o operație uzuală în statistică și aplicații pentru procesarea datelor
- Găsirea valorii medii este un caz particular de selecție: găsirea celui de-al k-lea mic element dintr-o listă
- Selecția are multe aplicații în procesarea datelor experimentale
- O soluție am văzut deja: TopM
- O altă soluție este să sortăm lista de elemente și să luăm a k-a valoare (soluție linearitmică)
- Dacă k este mic sau mare soluția este simplă
- Dacă k ≈ N/2 atunci avem o soluție liniară bazată pe partiționarea QuickSort

Algoritmi de sortare – aplicații

- **Găsirea valorii medii** - soluție liniară

```
public static T select<T>(T[] a, int k) where T: IComparable<T> {
        Util.shuffle(a);
        int lo = 0, hi = a.Length - 1;
        while (hi > lo){
            int j = partition(a, lo, hi);
            if (j == k) return a[k];
            else if (j > k) hi = j - 1;
            else if (j < k) lo = j + 1;
        }
        return a[k];
    }
```

Algoritmi de sortare – aplicații

- Sortarea este o operație omniprezentă în orice aplicație software
  - Biblioteci multimedia sunt sortate după numele artistului
  - Email-urile sunt sortate după data primirii
  - Imaginile sunt sortate după dată
  - Universitățile sortează studenții după nume, medie, ID
  - Oamenii de știință sortează datele experimentale după diverse criterii pentru a permite realizarea de simulări ale lumii naturale
- E greu de găsit o aplicație computațională care să nu utilizeze sortare
-

Algoritmi de sortare – aplicații

- Aplicații comerciale
  - La ora actuală există foarte multă informație în diverse baze de date comerciale
  - Organizațiile guvernamentale, instituțiile financiare, companiile comerciale organizează mare parte din această informație prin sortare
  - Conturi sortate după nume sau număr, tranzacții sortate după dată sau sumă, mesaje scrise sortate după adresă sau cod poștal, fișiere sortate după nume sau dată
  - Toate aceste date implică un algoritm de sortare
  - Datele sunt stocate în baze de date și sortate după mai multe chei pentru căutare mai ușoară, informația nouă este sortată după toate cheile și interclasată cu informația existentă în baza de date pentru fiecare cheie în parte
  - Bazele de date conțin milioane/miliarde de înregistrai care nu ar putea fi procesate fără algoritmi de sortare linearitmici

Algoritmi de sortare – aplicații

- Căutarea
  - Datele sortate facilitează operația de căutare (așa cum am văzut la căutarea binară)
  - Se poate răspunde și la alte tipuri de interogări
    - Câte elemente sunt mai mici decât o cheie dată?
    - Câte elemente se găsesc într-un anumit interval?
  - Vom discuta soluții la astfel de probleme
  - Vom discuta soluții de inserare și ștergere a unor elemente în listă în timp linearitmic

Algoritmi de sortare – aplicații

- Cercetări operaționale
  - Domeniu care se ocupă cu crearea și aplicarea de modele matematice pentru a rezolva diverse problem și a lua decizii
  - O problemă de programare: se dau N job-uri și jobul j se termină în timp t_j; se cere să programăm job-urile la execuție în așa fel încât să maximizăm satisfacția clientlui. Problema se poate rezolva prin regula *cel mai scurt timp de procesare primul*. Pentru aceasta joburile trebuie sortate crescător după timpul de execuție
  - O problema de *load-balancing*: se dau M procesoare identice și N job-uri care trebuie executate. Se cere să se programeze cele N job-uri pe cele M procesoare în așa fel încât ultimul job care se execută să se termina cât mai repede. Problemă NP-hard prin urmare nu vom găsi o soluție de calcul a unei programări optime. Putem aplica regula *cel mai lung timp de procesare primul*.

Algoritmi de sortare – aplicații

- Simulări *event-driven*
  - Multe aplicații științifice implică simulări
  - Scopul este modelarea unui aspect al lumii reale pentru a-l înțelege mai bine
  - Putem crea modele computaționale pe lângă modelele matematice
  - Utilizarea unor algoritmi eficienți pentru aceste simulări poate face diferența între realizarea lor într-un timp rezonabil și a nu le putea realiza deloc

Algoritmi de sortare – aplicații

- Calcule numerice
  - În calcule științifice e nevoie de acuratețe a rezultatelor
  - Acuratețea este importantă atunci când se fac milioane de operații cu valori de tip aproximative de tip float/double
  - Pentru controlul acurateței se pot folosi cozi cu prioritate și sortare
  - Exemplu: pentru a realiza integrare numerică (cuadratură) pe un interval pentru a estima aria unei suprafețe de sub o curbă putem menține o coadă cu prioritate cu estimări de acuratețe pentru o mulțime de subintervale care compun intervalul. Se elimină intervalul cu acuratețea cea mai mică, se împarte în două intervale (pentru a obține acuratețe mai bună) și cele două intervale se pun înapoi în coada cu prioritate. Se continuă astfel până când se obține acuratețea dorită

Algoritmi de căutare

- Tehnicile moderne de calcul și Internet-ul ne pun la dispoziție o cantitate foarte mare de informație
- Posibilitatea de a căuta în mod eficient în această informație este esențial pentru procesarea ei
- Vom descrie algoritmi de căutare clasici care s-au dovedit utili în numeroase aplicații de-a lungul anilor
- Fără acești algoritmi nu ar fi fost posibilă dezvoltarea infrastructurii computaționale pe care o avem azi la dispoziție astăzi

Algoritmi de căutare

- Tabela de simboluri = mecanism abstract pentru salvarea informației (o valoare) pe care o putem căuta și obține ulterior pe baza unei chei
- Natura cheilor și a valorilor depinde de aplicație
- Numărul cheilor și a valorilor poate fi foarte mare prin urmare implementarea unei tabele de simboluri eficientă este o importantă provocare computațională
- Tabela de simboluri se mai numește *dicționar*, prin analogie cu sistemele ce oferă definiții pentru cuvinte ordonate alfabetic
- În DEX cheia este un cuvânt iar valoarea este dată de definiția/definițiile acelui cuvânt împreună cu pronunțarea fonetic (și eventual etimologia, declinări, sinonime etc.)

Algoritmi de căutare

- Tabelele de simboluri se mai numesc indecși prin analogie cu sistemul de index de la sfârșitul unei cărți care oferă acces la termeni prin enumerarea lor în ordine alfabetică
- Într-un index de carte cheile sunt termenii iar valorile asociate unei chei sunt paginile cărții pe care se regăsește cheia (termenul)
- Vom discuta
  - implementări clasice/elementare pentru tabele de simboluri
  - Structuri de date clasice ce permit implementarea eficientă a tabelelor simbolice (arbori binari de căutare, arbori roșu-negru, tabele hash)

Algoritmi de căutare – tabele de simboluri

- Tabela de simboluri asociază o valoare cu o cheie
- Programatorul client poate *insera* o pereche (cheie, valoare) în tabela de simboluri
- Ulterior poate *căuta* valoarea asociată unei chei printre toate perechile (cheie, valoare) care au fost adăugate în colecție
- Pentru a implementa o tabelă de simboluri trebuie să definim o structură de date în care să păstrăm informația și să specificăm algoritmi pentru operațiile de inserare, căutare și alte operații pentru crearea și manipularea structurii de date

Algoritmi de căutare – tabele de simboluri

- Aplicații ale tabelelor de simboluri

Aplicație	Ce se caută	Cheia	Valoarea
Dicționar	Definiție	Cuvânt	Definiție
Index de carte	Paginile relevante	Termen	Lista cu numere de pagină
Fișiere partajate	Melodii pentru download	Numele melodiei	ID calculator
Gestiune conturi	Procesarea tranzacțiilor	Număr cont	Detalii tranzacții
Căutarea web	Pagini web	Cuvinte cheie	Lista numelor paginilor
Compilator	Tip și valoare	Nume variabilă	Tip și valoare

Algoritmi de căutare – tabele de simboluri

- API pentru tabela de simboluri (oferă un contract între client și implementare)

```
public ST() // crearea unei tabele de simboluri
// adaugarea perechii (cheie, valoare) in tabela; eliminarea cheii daca value este null
public void put(Key key, Value value)
// obtine valoarea asociata cheii; daca in tabela nu exista cheia se intoarce null
public Value get(Key key)
public void delete(Key key) // sterge din tabela cheia (si valoarea asociata)
public bool contains(Key key) // exista in tabela o valoare asociata cheii?
public bool isEmpty() //  tabela este goala?
public int size() // numarul de perechi (cheie, valoare) din tabela
public IEnumerable<Key> keys() // toate cheile din tabela
```

Algoritmi de căutare – tabele de simboluri

- Vom specifica metodele fără a specifica tipul elementelor procesate folosind generics
- Atât tipul cheii cât și al valorii sunt parametri de tip care trebuie specificați explicit
- Nu este specificat faptul că cheile sunt comparabile
- Dacă cheile sunt comparabile atunci se pot adăuga numeroase alte metode în API

Algoritmi de căutare – tabele de simboluri

- Convenții
  - Există o singură valoare asociată cu o cheie (nu avem chei duplicat)
  - Când se inserează o pereche (cheie, valoare) și cheia există deja în tablă noua valoare o înlocuiește pe cea veche
- Aceste convenții definesc abstractizarea numită *tablou asociativ*, în care ne putem gândi la o tabelă de simboluri ca și la un vector în care cheile sunt indici iar valorile sunt elementele tabloului
- În cazul vectorilor indecșii sunt de tip întreg (pe accesare rapidă) pe când în cazul tablourilor asociative sunt de tip arbitrar
- Unele limbaje de programare oferă suport direct pentru tablouri asociative cu o sintaxă de tipul st[key] unde key este de tip arbitrar

Algoritmi de căutare – tabele de simboluri

- Cheile nu trebuie sa fie null. Utilizarea cheilor null va avea ca rezultat excepții la execuție
- Nici o cheie nu poate fi asociată cu valoarea null. În legătură cu faptul că get() întoarce null dacă cheia nu se află în tabelă. Consecințe:
  - Astfel putem testa dacă există o valoare asociată unei chei prin get() == null
  - Putem șterge o pereche (cheie, valoare) din tabelă prin put(key, null)
- Ștergerea într-o tabelă simbolică se poate aborda prin două strategii
  - Ștergere *lazy*: asociem null unei chei și eventual eliminăm astfel de chei ulterior
  - Ștergere *eager*: cheia este eliminată imediat din tabelă

Algoritmi de căutare – tabele de simboluri

- Implementări implicite

Metoda	Implementare implicită
void delete(Key key)	put(key, null)
bool contains(Key key)	return get(key) != null
bool isEmpty()	return size() == 0

Algoritmi de căutare – tabele de simboluri

- Iterare – se poate realiza prin implementarea interfeței `IEnumerable` la fel cum am făcut pentru Stack, Queue
- Aici vom oferi o metodă `keys()` care întoarce un obiect de tip `IEnumerable<Key>`
- Egalitatea cheilor – se bazează pe conceptul de egalitate de obiecte. Orice obiect moștenește metoda `Equals()`
- Pentru multe clase ne putem baza pe implementarea implicită a metodei Equals().
- Pentru clasele noastre va trebui să suprascriem (override) metoda `Equals()`
- E indicat să facem cheile imutabile pentru că altfel nu se poate garanta consistența

Algoritmi de căutare – tabele de simboluri ordonate

- De regulă cheile dintr-o tabela simbolică sunt obiecte ce implementează interfața `IComparable`.
- Putem folosi metoda `a.CompareTo(b)` pentru a compara cheile `a` și `b`.
- O serie de implementări pentru tabele de simboluri pot profita de acest aspect pentru a implementa operațiile `put()` și `get()` în mod eficient
- Atunci când cheile sunt păstrate în ordine putem extinde API-ul pentru a defini o serie de operații naturale și utile (care chei sunt într-un anumit interval, care este cea mai mică/mare cheie care satisface un anumit predicat etc.)

Algoritmi de căutare – tabele de simboluri ordonate – API pentru o tabelă simbolică generică cu cheile ordonate

```
class BinarySearchST<Key, Value>
    where Key: IComparable<Key>
```

```
public BinarySearchST()
public BinarySearchST(int capacity)
public bool contains(Key key)
public int size()
public bool isEmpty()
public Value get(Key key)
public int rank(Key key)
public void put(Key key, Value val)
```

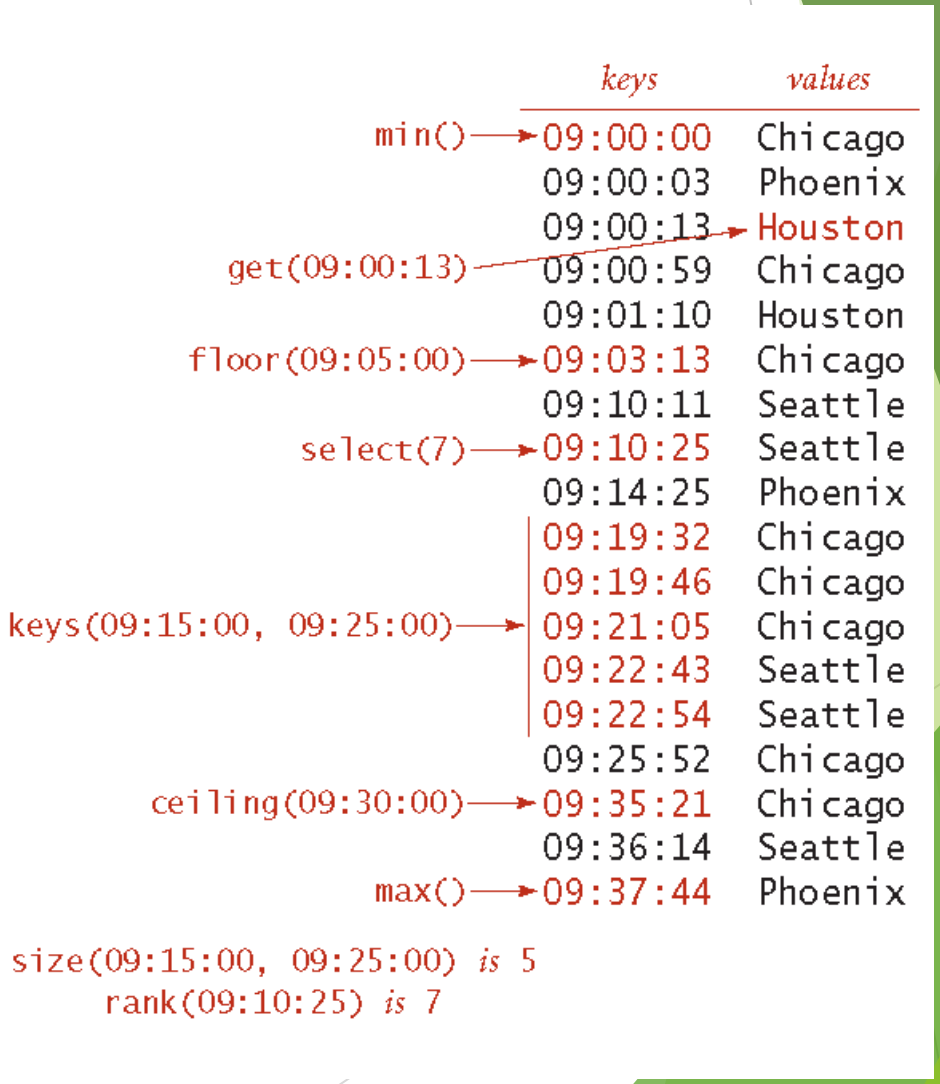
```
public void delete(Key key)
public void deleteMin()
public void deleteMax()
public Key min()
public Key max()
public Key select(int k)
public Key floor(Key key)
public Key ceiling(Key key)
public int size(Key lo, Key hi)
public IEnumerable<Key> Keys()
public IEnumerable<Key> Keys(Key lo, Key hi)
```

Algoritmi de căutare – tabele de simboluri ordonate

- Minim și maxim – operații naturale pentru o mulțime de chei ordonate
- Aceste operații le-am întâlnit și în cazul cozilor cu prioritate
- În tabelele simbolice avem și operațiile de ștergere a minimului și a maximului (și a valorilor asociate)
- Cu aceste operații tabela simbolică poate funcționa la fel ca și clasa `IndexMinPQ()`
- Diferența constă în următoarele:
  - În tabela simbolică nu putem avea chei egale
  - Tabela simbolică suportă mai multe operații

Algoritmi de căutare – tabele de simboluri ordonate

- Operații pe tabele de simboluri ordonate
- `floor(x)` cea mai mare cheie mai mică sau egală decât x
- `ceiling(x)` cea mai mică cheie mai mare sau egală decât x
- `rank(x)` numărul de chei mai mici decât x
- select(x) cheia cu rank x
- `i == rank(select(i)` pt. orice i de la 0 la size()-1
- `key == select(rank(key)`



Algoritmi de căutare – tabele de simboluri ordonate

- Operații pe tabele de simboluri ordonate
- `size(lo, hi)` – câte chei sunt într-un anumit interval
- `Keys(lo, hi)` – care sunt cheile dintr-un anumit interval
- Aceste două operații
  - Utile în multe aplicații cum ar fi baze de date
  - Motivul principal pentru care tabelele de simboluri sunt utilizate foarte des în practică
- Când o metodă trebuie să returneze o cheie dar nu există nici o cheie care să se potrivească cerinței metoda poate întoarce `null` sau poate lansa excepție

Algoritmi de căutare – tabele de simboluri ordonate

- Metoda `CompareTo()` a interfeței `IComparable` trebuie să fie consistentă cu metoda `Equals()` suprascrisă
- `a.CompareTo(b) == 0` trebuie să aibă același rezultat ca și `a.equals(b)`
- În implementări vom folosi metoda `CompareTo()` pentru a evita ambiguitățile
- Multe clase din biblioteca standard implementează interfața `IComparable` și oferă metoda `CompareTo()`
- Pentru clasele noastre va trebui să facem noi aceste implementări

Algoritmi de căutare – tabele de simboluri ordonate

- Modelul de cost:
- Indiferent că folosim `Equals()` (pentru tabele de simboluri care nu implementează `IComparable`) sau `CompareTo()` *comparare* se referă la operația la compararea unei valori din tablă cu o cheie de căutare.
- Operația de comparare de regulă este în bucla internă
- În studiul implementării tabelelor de simboluri vom număra comparațiile efectuate
- Atunci când comparația nu se face în bucla internă se vor număra accesările la elementele vectorului

Algoritmi de căutare – tabele de simboluri

- Client de test
- O secvență de stringuri din intrare este introdusă într-o tabelă simbolică
- Fiecărui string i se asociază o valoarea numerică în ordine crescătoare
- Se afișează tabela simbolică
- Secvența de stringuri S,E,A,R,C,H,E, X,A,M,P,L,E
- S – 0, E – 12, A – 8, R - 3 etc.
- Cheia se asociază cu cea mai recentă valoare

```
static void Main(string[] args)
        {
            ST<String, int?> st = new ST<String, int?>();
            String key = Console.ReadLine();
            for (int i = 0; key != null; i++)
            {
                st.put(key, i);
                key = Console.ReadLine();
            }
            foreach (var item in st.keys())
            {
                Console.WriteLine("{0} {1}", item, st.get(item));
            }
        }
```

Algoritmi de căutare – tabele de simboluri

- Client pentru tabela de simboluri – FrequencyCounter.cs
- Găsește numărul de apariții pentru fiecare cuvânt distinct care are un număr minim de litere dat ca si argument în linia de comandă
- Găsește cuvântul cu cea mai mare frecvență de apariție
- Exemplu de client pentru un *dicționar*
- Dă răspuns la o întrebare simplă: care este cuvântul (a cărui lungime este mai mare decât o valoare dată) care apare cel mai frecvent într-un text
- Ne interesează performanța acestei operații atunci când rulăm algoritmul cu diverse date de intrare

Algoritmi de căutare – tabele de simboluri

- Câteva exemple de date de intrare pentru FrequencyCounter

	tinyTale.txt		tale.txt		leipzig1M.txt	
	words	**distinct**	**words**	**distinct**	**words**	**distinct**
all words	60	20	135,635	10,679	21,191,455	534,580
at least 8 letters	3	3	14,350	5,737	4,239,597	299,593
at least 10 letters	2	2	4,582	2,260	1,610,829	165,555

Algoritmi de căutare – tabele de simboluri

- Se poate crea o implementare pentru o TS ce poate gestiona un număr foarte mare de operații `get()` pe tabelă mare care la rândul ei a fost creată printr-un număr foarte mare de operații `get()` și `put()` intercalate?
- FrequencyCounter este un surogat pentru o situație foarte comună:
  - Operațiile de inserare și căutare sunt intercalate
  - Numărul de chei distincte nu este mic
  - Ne putem aștepta la mult mai multe operații de căutare decât inserare
  - Operațiile de inserare și căutare nu sunt aleatorii dar sunt nepredictibile
- Obiectivul nostru este să realizăm o implementare care să facă fezabile aceste operații

Algoritmi de căutare – tabele de simboluri - Implementări

- Căutare secvențială într-o listă înlănțuită St.cs, StClient.cs
- Pentru a implementa operație `get()` parcugem lista și dacă găsim cheia returnăm valoarea asociată altfel returnăm `null`.

```
public Value get(Key key) {
            for (Node x = first; x != null ; x = x.next) {
                if (key.Equals(x.key))
                    return x.val;
            }
            return default(Value);
        }
```

Algoritmi de căutare – tabele de simboluri - Implementări

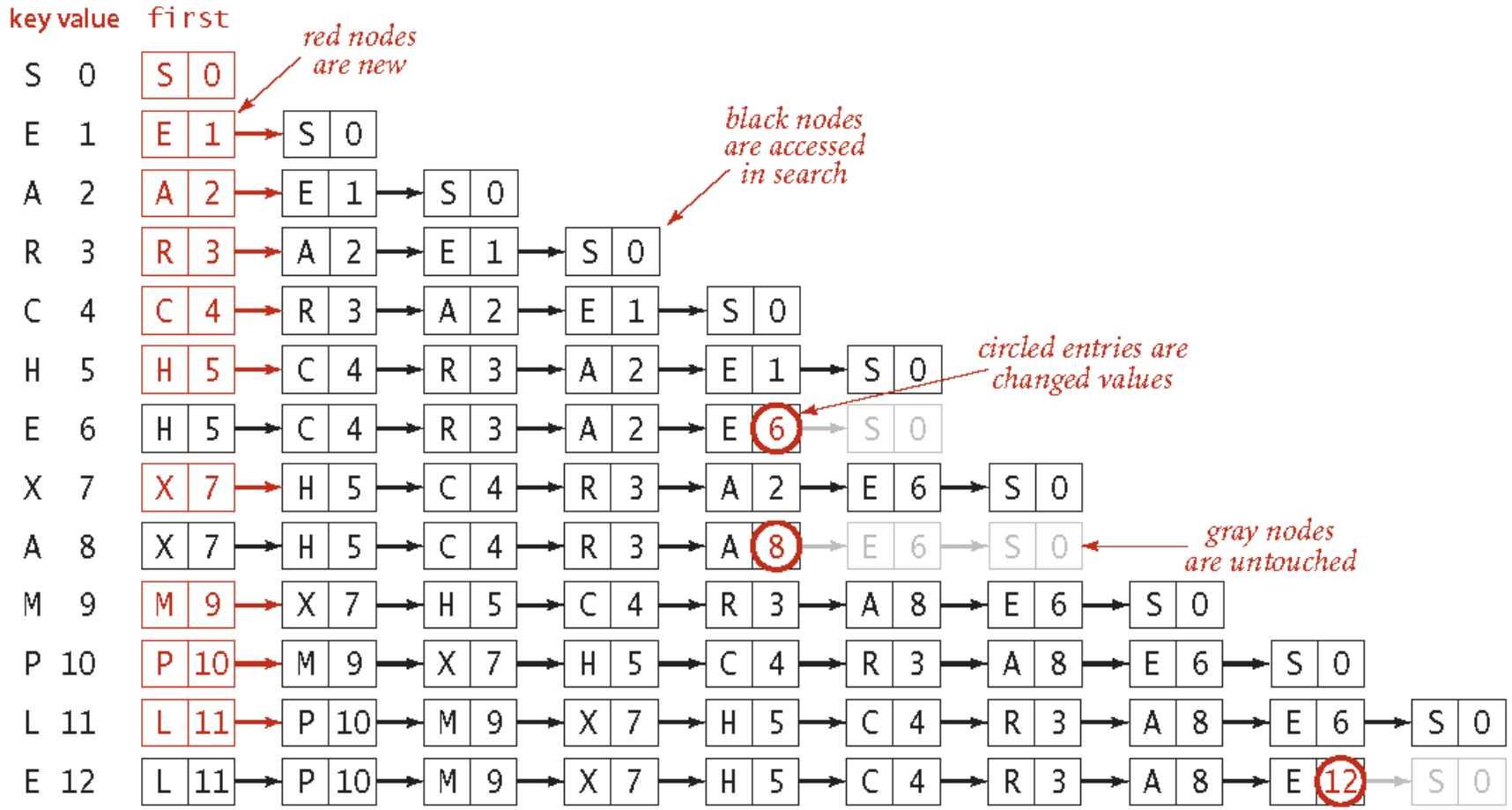
- Pentru a implementa operația put() parcurgem din nou lista secvențial și
  - Dacă găsim cheia actualizăm valoare cu noua valoare pe care o dăm metodei put()
  - Dacă nu găsim cheia în tabelă atunci se creează un nou nod (cheie, valoare) care se inserează la începutul listei

```
public void put(Key key, Value value) {
            if (value == null) {
                delete(key);  return;
            }
            for (Node x = first; x != null; x = x.next)
                if (key.Equals(x.key))  {
                    x.val = value; return;
                }
            first = new Node(key, value, first);
            N++;
        }
```

Algoritmi de căutare – tabele de simboluri - Implementări

```
class ST<Key, Value>  {
        private int N; // numarul de perechi (cheie, valoare) din colectie
        private Node first; // lista inalantuita de perechi (cheie, valoare)
        private class Node   {
            public Key key;
            public Value val;
            public Node next;
        }
        public Node(Key key, Value val, Node next)       {
                this.key = key;    this.val = val;     this.next = next;
            }
        }
        ...
}
```

Algoritmi de căutare – tabele de simboluri - Implementări



Algoritmi de căutare – tabele de simboluri – Implementări

- Implementarea bazată pe listă înlănțuită poate trata liste foarte lungi?
- Analizarea