## 2. Структури на податоци

Во овој предмет се изучуваат основни структури на податоци и алгоритми за изведување на операции со податоците сместени во структурите. Структури на податоци се организации на податоци во работната меморија или дискот на компјутерот. Во податочни структури спаѓаат полиња, поврзани листи, магацини, бинарни дрва, хеш табели и други. Алгоритмите извршуваат одредени операции со податоците. На пример, алгоритмите може да ги пребаруваат структурите на податоци или да ги сортираат податоците.

Структурите на податоци се користат за сместување на податоци од различни домени. На пример, лични податоци, податоци од финансиски трансакции, податоци за резервирање на авионски билети и др. При претставување на податоците треба да се има во предвид колкава количина на податоци треба да се меморира и да се изберат структури кои овозможуваат ефикасно вметнување, бришење или наоѓање на соодветен податок. Во табелата 2.1 е даден преглед на структури на податоци заедно со нивните предности и недостатоци.

Табела 2.1 Карактеристики на структурите на податоци

|  |  |  |
| --- | --- | --- |
| **Податочна структура** | **предности** | **недостатоци** |
| Поле | Брзо вметнување, многу брз пристап ако е познат индексот на податокот | Бавно пребарување, бавно бришење, фиксна големина. |
| Подредено поле | Побрзо пребарување во споредна со несортирано поле. | Бавно вметнување и бришење, фиксна големина. |
| Магацин (stack) | Обезбедува последен внесен, прв избришан пристап. | Бавен пристап до другите податоци. |
| Ред (queue) | Обезбедува прв внесен прв избришан пристап. | Бавен пристап до другите податоци. |
| Поврзана листа | Брзо вметнување, брзо бришење. | Бавно пребарување. |
| Бинарно дрво | Брзо пребарување,  вметнување, бришење (ако дрвото останува балансирано) | Алгоритамот за бришење е комплексен. |
| Балансирано дрво  (red-black tree) | Брзо пребарување, вметнување, бришење. Дрвото е секогаш балансирано. | Комплексно. |
| 2-3-4 дрво | Брзо пребарување, вметнување, бришење. Дрвото е секогаш балансирано. Слични дрва добри за сместување на диск. | Комплексно. |
| Хеш табела | Многу брз пристап ако е познат клучот. Брзо вметнување. | Бавно бришење, бавен пристап ако не е познат клучот, неефикасно користење на меморија. |
| Хип (heap) | Брзо вметнување, бришење,  пристап до најголемиот податок. | Бавен пристап до други податоци. |
| Граф | Моделира реални ситуации. | Некои алгоритми се бавни и комплексни. |

Освен полињата сите други структури во табелата 2.1 се нарекуваат апстрактни податочни типови.

### 2.1 Полиња

Полињата се најчесто користени податочни структури вградени во повеќето програмски јазици. Во ова поглавје се воведуваат полиња како податочни структури во Јава. Се воведуваат и подредени полиња (во растечки или опаѓачки редослед на елементите) кои овозможуваат брзо пребарување наречено бинарно пребарување.

Ако полето има N елементи алгоритмот за пребарување треба просечно да направи N/2 проверки. Елементите кои се поблиску до почетокот на полето ќе бидат пронајдени побрзо, а оние на крајот на полето побавно. Во најлош случај, ако бараниот елемент е последен во полето треба да се направат N споредби. Времето на извршување на алгоритмот за пребарување е пропорционално на бројот на елементи во полето и изнесува просечно N/2 чекори.

За да се избрише елемент тој треба прво да се пронајде. Откога елементот се избрише треба да се поместат преостанатите елементи до крајот на полето за едно место во лево. Бришењето се извршува за просечно N/2 чекори за пребарување и потоа просечно N/2 чекори за поместување на елементите, односно пополнување на празните места. Според тоа вкупното време на бришењето е N. Ако се дозволени повторувања на елементи алгоритмите за пребарување и бришење стануваат покомплексни. Пребарувањето на елемент во поле со дупликати се извршува за N чекори бидејќи треба да се изминат сите елементи на полето за да се најдат сите појавувања на бараниот податок. При бришење на податок од поле каде се дозволени дупликати ако треба да се избрише само едно појавување на податокот потребни се просечно N/2 споредби и N/2 поместувања. Ако треба да се избришат сите појавувања на податокот потребни се N споредби и поместување на повеќе од N/2 елементи. Вметнувањето на елемент се извршува во еден чекор, без разлика дали се или не се дозволени повторувања на елементите во полето. Табелата 2.2 го прикажува бројот на споредби и поместувања за основните операции со полиња кога се и не се дозволени повторувања на елементите во полето.

Табела 2.2 Број на споредби и поместувања при операциите вметнување, пребарување и бришење кај полиња

|  |  |  |
| --- | --- | --- |
| **Операција со полиња** | **Не се дозволени повторувања** | **Дозволени се повторувања** |
| пребарување | N/2 споредби | N споредби |
| вметнување | Нема споредби, еден чекор | Нема споредби, еден чекор |
| бришење | N/2 споредби,  N/2 поместувања | N споредби,  повеќе од N/2 поместувања |

### 2.2 Полиња во Јава

Како што е кажано во поглавјето 1 кое ги опишува основите на програмирањето во Јава, има два типа на податоци: примитивни типови (како int и double) и објекти. Во многу програнски јазици (како C++) полињата се примитивни типови, но во Јава се третираат како објекти. Според тоа мора да се користи операторот new за да се креира поле:

int[] intArray; // defines a reference to an array

intArray = new int[100]; // creates the array, and

// sets intArray to refer to it

Горните наредби може да се реализираат со една наредба:

int[] intArray = new int[100];

Алтернативна синтакса за операторот [] која му кажува на компајлерот дека се дефинира поле а не обична променлива е следната

int intArray[] = new int[100]; // alternative syntax

Бидејќи полето е објект неговото име intArray е референца на поле, а не самото поле. Полето се чува на некоја адреса во меморија а intArray ја чува таа адреса.

Полињата имаат атрибут должина length со кој се одредува бројот на елементи на полето:

int arrayLength = intArray.length; // find array size

Како и кај повеќето програмски јазици не може да се промени должината на полето по неговото креирање. До елемент на полето се пристапува со користење на индексот во средни загради, како кај другите програмски јазици:

temp = intArray[3]; // get contents of fourth element of array

intArray[7] = 66; // insert 66 into the eighth cell

Во Јава (исто како во C и C++) елементите на полето имаат индекс кој започнува од 0. Така поле со 10 елементи ќе има индекси за елементите од 0 до 9. Не е дозволено надминување на опсегот на полињата. Во случај на надминување на опсегот се генерира грешка при извршување на програмата Array Index Out of Bounds.

Ако не се специфицира поинаку полето од цели броеви (integers) автоматски се иницијализира со 0. Може да се креира и поле од објекти кое почетно се иницијализира со null објекти:

autoData[] carArray = new autoData[4000];

Ако се пристапи до поле кое содржи null објекти се добива грешка при извршување Null Pointer Assignment.

Поле со елементи кои се примитивни типови може да се иницијализира на следниот начин:

int[] intArray = { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 };

Горниот израз декларира референца и го користи new за да креира поле. Големината на полето е определена со бројот на вредности во иницијализирачката листа.

Првата програма која се креира како пример за работа со полиња е процедурална верзија а втората програма користи објектно ориентиран приод.

// array.java

// demonstrates Java arrays

// to run this program: C>java arrayApp

////////////////////////////////////////////////////////////////

class ArrayApp

{

public static void main(String[] args)

{

long[] arr; // reference to array

arr = new long[100]; // make array

int nElems = 0; // number of items

int j; // loop counter

long searchKey; // key of item to search for

//--------------------------------------------------------------

arr[0] = 77; // insert 10 items

arr[1] = 99;

arr[2] = 44;

arr[3] = 55;

arr[4] = 22;

arr[5] = 88;

arr[6] = 11;

arr[7] = 00;

arr[8] = 66;

arr[9] = 33;

nElems = 10; // now 10 items in array

//--------------------------------------------------------------

for(j=0; j<nElems; j++) // display items

System.out.print(arr[j] + “ “);

System.out.println(“”);

//--------------------------------------------------------------

searchKey = 66; // find item with key 66

for(j=0; j<nElems; j++) // for each element,

if(arr[j] == searchKey) // found item?

break; // yes, exit before end

if(j == nElems) // at the end?

System.out.println(“Can’t find “ + searchKey); // yes

else

System.out.println(“Found “ + searchKey); // no

//--------------------------------------------------------------

searchKey = 55; // delete item with key 55

for(j=0; j<nElems; j++) // look for it

if(arr[j] == searchKey)

break;

for(int k=j; k<nElems-1; k++) // move higher ones down

arr[k] = arr[k+1];

nElems--; // decrement size

//--------------------------------------------------------------

for(j=0; j<nElems; j++) // display items

System.out.print( arr[j] + “ “);

System.out.println(“”);

} // end main()

} // end class ArrayApp

Во горната програма се креира поле наречено arr, во него се сместуваат 10 елементи, се пребарува за елементот со вредност 66, се печатат елементите на полето, се брише елементот со вредност 55 и се печатат останатите 9 елементи. Излезот од програмата е слениот:

77 99 44 55 22 88 11 0 66 33

Found 66

77 99 44 22 88 11 0 66 33

Горната програма може да се напише со користење на повеќе класи. Една класа е предвидена за сместување на податоци, а делот од програмата кој ги користи податоците е втора класа. Поделбата се прави за да се овозможи поголема функционалност и полесно разбирање и одржување на програмскиот код.

Полето се енкапсулира во класа LowArray. Потоа се креира класа LowArrayApp која содржи методи за пристап до полето.

// lowArray.java

// demonstrates array class with low-level interface

// to run this program: C>java LowArrayApp

////////////////////////////////////////////////////////////////

class LowArray

{

private long[] a; // ref to array a

//--------------------------------------------------------------

public LowArray(int size) // constructor

{ a = new long[size]; } // create array

//--------------------------------------------------------------

public void setElem(int index, long value) // set value

{ a[index] = value; }

//--------------------------------------------------------------

public long getElem(int index) // get value

{ return a[index]; }

//--------------------------------------------------------------

} // end class LowArray

////////////////////////////////////////////////////////////////

class LowArrayApp

{

public static void main(String[] args)

{

LowArray arr; // reference

arr = new LowArray(100); // create LowArray object

int nElems = 0; // number of items in array

int j; // loop variable

arr.setElem(0, 77); // insert 10 items

arr.setElem(1, 99);

arr.setElem(2, 44);

arr.setElem(3, 55);

arr.setElem(4, 22);

arr.setElem(5, 88);

arr.setElem(6, 11);

arr.setElem(7, 00);

arr.setElem(8, 66);

arr.setElem(9, 33);

nElems = 10; // now 10 items in array

for(j=0; j<nElems; j++) // display items

System.out.print(arr.getElem(j) + “ “);

System.out.println(“”);

int searchKey = 26; // search for data item

for(j=0; j<nElems; j++) // for each element,

if(arr.getElem(j) == searchKey) // found item?

break;

if(j == nElems) // no

System.out.println(“Can’t find “ + searchKey);

else // yes

System.out.println(“Found “ + searchKey);

// delete value 55

for(j=0; j<nElems; j++) // look for it

if(arr.getElem(j) == 55)

break;

for(int k=j; k<nElems; k++) // higher ones down

arr.setElem(k, arr.getElem(k+1) );

nElems--; // decrement size

for(j=0; j<nElems; j++) // display items

System.out.print( arr.getElem(j) + “ “);

System.out.println(“”);

} // end main()

} // end class LowArrayApp

////////////////////////////////////////////////////////////////

Излезот од програмата е сличен на претходниот, освен што се пребарува полето за непостоечка вредност 26, пред да се избрише елемент со вредност 55:

77 99 44 55 22 88 11 0 66 33

Can’t find 26

77 99 44 22 88 11 0 66 33

Следно подобрување на програмата е користење подобар интерфејс за класата која го чува полето HighArray. Наместо методите setElem() и getElem() кои работат со индексите на елементите, сега се користат insert(), find() и delete() на кои не треба да им се проследи индекс на елемент бидејќи класата HighArray ракува со индексите. Класата корисник HighArrayApp се грижи за тоа што а не како треба да се вметне, најде или избрише.

// highArray.java

// demonstrates array class with high-level interface

// to run this program: C>java HighArrayApp

////////////////////////////////////////////////////////////////

class HighArray

{

private long[] a; // ref to array a

private int nElems; // number of data items

//-----------------------------------------------------------

public HighArray(int max) // constructor

{

a = new long[max]; // create the array

nElems = 0; // no items yet

}

//-----------------------------------------------------------

public boolean find(long searchKey)

{ // find specified value

int j;

for(j=0; j<nElems; j++) // for each element,

if(a[j] == searchKey) // found item?

break; // exit loop before end

if(j == nElems) // gone to end?

return false; // yes, can’t find it

else

return true; // no, found it

} // end find()

//-----------------------------------------------------------

public void insert(long value) // put element into array

{

a[nElems] = value; // insert it

nElems++; // increment size

}

//-----------------------------------------------------------

public boolean delete(long value)

{

int j;

for(j=0; j<nElems; j++) // look for it

if( value == a[j] )

break;

if(j==nElems) // can’t find it

return false;

else // found it

{

for(int k=j; k<nElems; k++) // move higher ones down

a[k] = a[k+1];

nElems--; // decrement size

return true;

}

} // end delete()

//-----------------------------------------------------------

public void display() // displays array contents

{

for(int j=0; j<nElems; j++) // for each element,

System.out.print(a[j] + “ “); // display it

System.out.println(“”);

}

//-----------------------------------------------------------

} // end class HighArray

////////////////////////////////////////////////////////////////

class HighArrayApp

{

public static void main(String[] args)

{

int maxSize = 100; // array size

HighArray arr; // reference to array

arr = new HighArray(maxSize); // create the array

arr.insert(77); // insert 10 items

arr.insert(99);

arr.insert(44);

arr.insert(55);

arr.insert(22);

arr.insert(88);

arr.insert(11);

arr.insert(00);

arr.insert(66);

arr.insert(33);

arr.display(); // display items

int searchKey = 35; // search for item

if( arr.find(searchKey) )

System.out.println(“Found “ + searchKey);

else

System.out.println(“Can’t find “ + searchKey);

arr.delete(00); // delete 3 items

arr.delete(55);

arr.delete(99);

arr.display(); // display items again

} // end main()

} // end class HighArrayApp

////////////////////////////////////////////////////////////////

Излезот од програмата е следниот

77 99 44 55 22 88 11 0 66 33

Can’t find 35

77 44 22 88 11 66 33

Полето има 10 елементи и од него се бришат 3 елементи (0, 55 и 99).

### 2.3 Подредени полиња

За реализација на подредено поле се креира класата OrdArray. Со методот find() на оваа класа се изведува бинарно пребарување. Методот find() се претставува на следниот начин:

public int find(long searchKey)

{

int lowerBound = 0;

int upperBound = nElems-1;

int curIn;

while(true)

{

curIn = (lowerBound + upperBound ) / 2;

if(a[curIn]==searchKey)

return curIn; // found it

else if(lowerBound > upperBound)

return nElems; // can’t find it

else // divide range

{

if(a[curIn] < searchKey)

lowerBound = curIn + 1; // it’s in upper half

else

upperBound = curIn - 1; // it’s in lower half

} // end else divide range

} // end while

} // end find()

Методот започнува со поставување на вредности на променливите lowerBound и upperBound за првиот и последниот индекс во полето. Тоа се границите во кои ќе се бара елементот searchKey. Потоа во while циклусот тековниот индекс curIn се поставува на средината од опсегот за пребарување. Прво се проверува дали бараниот елемент се наоѓа на средината на опсегот. Потоа во секој циклус опсегот кој се пребарува се дели на половина. Ако опсегот веќе не може да се дели односно lowerBound е поголемо од upperBound пребарувањето завршува и се враќа индекс nElems, кој не е валиден индекс. Ако lowerBound е еднакво на upperBound во опсегот има само еден елемент и треба да се провери дали е бараниот елемент. Ако curIn не покажува на бараниот елемент и опсегот е поголем потребно е да се најде средниот елемент од тој опсег. Потоа се споредува средниот елемент со елементот кој се бара searchKey. Ако searchKey е поголем треба да се пребарува горната половина на опсегот а ако е помал долната. Сликата 2.1 го прикажува делењето на опсегот при бинарно пребарување:



**Слика 2.1** Делење на опсегот при бинарно пребарување

При внесување на елемент нема да се постигне значајно забрзување и во случај да се користи бинарно пребарување бидејќи просечно половина од елементите треба да се поместат. Истата забелешка се однесува и за бришење на елемент при што треба да се поместат елементите по избришаниот елемент бидејќи не се дозволени празни места.

Класата OrdArray користи метод size() кој враќа број на елементи тековно сместени во полето. Во продолжение е даден листинг на програмата која работи со подредени полиња:

// orderedArray.java

// demonstrates ordered array class

// to run this program: C>java OrderedApp

////////////////////////////////////////////////////////////////

class OrdArray

{

private long[] a; // ref to array a

private int nElems; // number of data items

//-----------------------------------------------------------

public OrdArray(int max) // constructor

{

a = new long[max]; // create array

nElems = 0;

}

//-----------------------------------------------------------

public int size()

{ return nElems; }

//-----------------------------------------------------------

public int find(long searchKey)

{

int lowerBound = 0;

int upperBound = nElems-1;

int curIn;

while(true)

{

curIn = (lowerBound + upperBound ) / 2;

if(a[curIn]==searchKey)

return curIn; // found it

else if(lowerBound > upperBound)

return nElems; // can’t find it

else // divide range

{

if(a[curIn] < searchKey)

lowerBound = curIn + 1; // it’s in upper half

else

upperBound = curIn - 1; // it’s in lower half

} // end else divide range

} // end while

} // end find()

//-----------------------------------------------------------

public void insert(long value) // put element into array

{

int j;

for(j=0; j<nElems; j++) // find where it goes

if(a[j] > value) // (linear search)

break;

for(int k=nElems; k>j; k--) // move bigger ones up

a[k] = a[k-1];

a[j] = value; // insert it

nElems++; // increment size

} // end insert()

//-----------------------------------------------------------

public boolean delete(long value)

{

int j = find(value);

if(j==nElems) // can’t find it

return false;

else // found it

{

for(int k=j; k<nElems; k++) // move bigger ones down

a[k] = a[k+1];

nElems--; // decrement size

return true;

}

} // end delete()

//-----------------------------------------------------------

public void display() // displays array contents

{

for(int j=0; j<nElems; j++) // for each element,

System.out.print(a[j] + “ “); // display it

System.out.println(“”);

}

//-----------------------------------------------------------

} // end class OrdArray

////////////////////////////////////////////////////////////////

class OrderedApp

{

public static void main(String[] args)

{

int maxSize = 100; // array size

OrdArray arr; // reference to array

arr = new OrdArray(maxSize); // create the array

arr.insert(77); // insert 10 items

arr.insert(99);

arr.insert(44);

arr.insert(55);

arr.insert(22);

arr.insert(88);

arr.insert(11);

arr.insert(00);

arr.insert(66);

arr.insert(33);

int searchKey = 55; // search for item

if( arr.find(searchKey) != arr.size() )

System.out.println(“Found “ + searchKey);

else

System.out.println(“Can’t find “ + searchKey);

arr.display(); // display items

arr.delete(00); // delete 3 items

arr.delete(55);

arr.delete(99);

arr.display(); // display items again

} // end main()

} // end class OrderedApp

#### 2.3.1 Предности на подредените полиња

Главна придобивка од подредувањето на полињата е тоа што пребарувањето е многу побрзо во споредба со полињата кои не се подредени. Недостаток кај подредените полиња е што вметнувањето на елемент трае подолго бидејќи сите елементи со поголем клуч треба да се поместат за да направат место. Бришењата се бавни и кај подредените и кај неподредените полиња бидејќи треба да се направат поместувања за да се пополни местото на избришаниот елемент. Според тоа подредените полиња треба да се користата кога има чести пребарувања на податоци а мал број на вместнувања и бришења. На пример, подредени полиња може да се користат при креирање на база на податоци за вработени. Вработување на нови работници или заминување на работници е ретко во споредба со пристапите до информациите за вработените. За чување податоци за производи во продавница подредените полиња не се соодветни заради честите вметнувања и бришења на производи кои ќе ја забавуваат работата на програмата.

#### 2.3.2 Предности на бинарното пребарување во однос на линеарното пребарување

Претходно е спомнато дека бинарното пребарување има значително поголема брзина од линеарното пребарување. На пример, за поле од 100 елементи потребни се 7 споредби за да се најде слог со одреден клуч, додека за поле од 1 000 000 000 елементи потребни се 30 споредби. Во табелата 2.3 е даден бројот на потребни споредби при бинарно пребарување на полиња со различен број на елементи.

Табела 2.3 Број на споредби при бинарно пребарување на полиња

|  |  |  |
| --- | --- | --- |
| Број на споредби s | Опсег на поле r | Опсегот на полето изразен како степен на бројот 2 () |
| 0 | 1 |  |
| 1 | 2 |  |
| 2 | 4 |  |
| 3 | 8 |  |
| 4 | 16 |  |
| 5 | 32 |  |
| 6 | 64 |  |
| 7 | 128 |  |
| 8 | 256 |  |
| 9 | 512 |  |
| 10 | 1024 |  |

Бројот на споредби во општ случај при бинарно пребарување на поле се пресметува по формулата s, каде s е бројот на споредби а r опсегот на полето, односно бројот на елементи на полето.

### 2.4 Полиња од објекти

Во досегашните примери во податочните структури полиња елементите се примитивни променливи од типот long. Вообичаено во реалните програми се работи со полиња од објекти. Објектите се комбинација од повеќе атрибути. На пример, слогот за вработен треба да содржи информација за име, презиме, возраст, матичен број, итн. Во следниот пример прикажана е работа со поле од објекти, од класата Person. Клуч на објектите е презимето кое е String објект. За споредба на стрингови се користи методот equals() а не операторот = =.

// classDataArray.java

// data items as class objects

// to run this program: C>java ClassDataApp

////////////////////////////////////////////////////////////////

class Person

{

private String lastName;

private String firstName;

private int age;

//--------------------------------------------------------------

public Person(String last, String first, int a)

{ // constructor

lastName = last;

firstName = first;

age = a;

}

//--------------------------------------------------------------

public void displayPerson()

{

System.out.print(“ Last name: “ + lastName);

System.out.print(“, First name: “ + firstName);

System.out.println(“, Age: “ + age);

}

//--------------------------------------------------------------

public String getLast() // get last name

{ return lastName; }

} // end class Person

////////////////////////////////////////////////////////////////

class ClassDataArray

{

private Person[] a; // reference to array

private int nElems; // number of data items

public ClassDataArray(int max) // constructor

{

a = new Person[max]; // create the array

nElems = 0; // no items yet

}

//--------------------------------------------------------------

public Person find(String searchName)

{ // find specified value

int j;

for(j=0; j<nElems; j++) // for each element,

if( a[j].getLast().equals(searchName) ) // found item?

break; // exit loop before end

if(j == nElems) // gone to end?

return null; // yes, can’t find it

else

return a[j]; // no, found it

} // end find()

//-------------------------------------------------------------

// put person into array

public void insert(String last, String first, int age)

{

a[nElems] = new Person(last, first, age);

nElems++; // increment size

}

//--------------------------------------------------------------

public boolean delete(String searchName)

{ // delete person from array

int j;

for(j=0; j<nElems; j++) // look for it

if( a[j].getLast().equals(searchName) )

break;

if(j==nElems) // can’t find it

return false;

else // found it

{

for(int k=j; k<nElems; k++) // shift down

a[k] = a[k+1];

nElems--; // decrement size

return true;

}

} // end delete()

//--------------------------------------------------------------

public void displayA() // displays array contents

{

for(int j=0; j<nElems; j++) // for each element,

a[j].displayPerson(); // display it

}

//--------------------------------------------------------------

} // end class ClassDataArray

////////////////////////////////////////////////////////////////

class ClassDataApp

{

public static void main(String[] args)

{

int maxSize = 100; // array size

ClassDataArray arr; // reference to array

arr = new ClassDataArray(maxSize); // create the array

// insert 10 items

arr.insert(“Evans”, “Patty”, 24);

arr.insert(“Smith”, “Lorraine”, 37);

arr.insert(“Yee”, “Tom”, 43);

arr.insert(“Adams”, “Henry”, 63);

arr.insert(“Hashimoto”, “Sato”, 21);

arr.insert(“Stimson”, “Henry”, 29);

arr.insert(“Velasquez”, “Jose”, 72);

arr.insert(“Lamarque”, “Henry”, 54);

arr.insert(“Vang”, “Minh”, 22);

arr.insert(“Creswell”, “Lucinda”, 18);

arr.displayA(); // display items

String searchKey = “Stimson”; // search for item

found=arr.find(searchKey);

if(found != null)

{

System.out.print(“Found “);

found.displayPerson();

}

else

System.out.println(“Can’t find “ + searchKey);

System.out.println(“Deleting Smith, Yee, and Creswell”);

arr.delete(“Smith”); // delete 3 items

arr.delete(“Yee”);

arr.delete(“Creswell”);

arr.displayA(); // display items again

} // end main()

} // end class ClassDataApp

////////////////////////////////////////////////////////////////

Излезот од програмата е:

Last name: Evans, First name: Patty, Age: 24

Last name: Smith, First name: Lorraine, Age: 37

Last name: Yee, First name: Tom, Age: 43

Last name: Adams, First name: Henry, Age: 63

Last name: Hashimoto, First name: Sato, Age: 21

Last name: Stimson, First name: Henry, Age: 29

Last name: Velasquez, First name: Jose, Age: 72

Last name: Lamarque, First name: Henry, Age: 54

Last name: Vang, First name: Minh, Age: 22

Last name: Creswell, First name: Lucinda, Age: 18

Found Last name: Stimson, First name: Henry, Age: 29

Deleting Smith, Yee, and Creswell

Last name: Evans, First name: Patty, Age: 24

Last name: Adams, First name: Henry, Age: 63

Last name: Hashimoto, First name: Sato, Age: 21

Last name: Stimson, First name: Henry, Age: 29

Last name: Velasquez, First name: Jose, Age: 72

Last name: Lamarque, First name: Henry, Age: 54

Last name: Vang, First name: Minh, Age: 22

Сериозна програма која го користи презимето како клуч треба да вклучува и механизам за справување со презимиња кои се повторуваат.

### 2.5 Ознака големо О

Ознаката големо О (Big O) се користи како мерка за ефикасноста на алгоритмите. При сподба на два алгоритма може да се каже дека „алгоритмот А е двапати побрз од алгоритмот Б“. Односот меѓу брзините на два алгоритма може да се менува со менување на бројот на елементи. На пример, ако се зголеми бројот на елементи за 50% може А да стане три пати побрз од Б. Затоа потребна е мерка која ја определува брзината на алгоритамот во зависност од бројот на елементи.

Времето за вметнување на елемент во поле Т е исто без разлика колкава е големината на полето бидејќи елементот се вметнува на крајот на полето, по последниот внесен елемент. Па може да се напише Т=К. Во реални услови времето на вметнување на елемент ќе зависи од брзината на микропроцесорот каде се извршува програмата, колку ефикасно компајлерот го генерирал програмскиот код и др. Константата К се определува според тие фактори.

При линеарно пребарување на поле времето е пропорционално на бројот на елементи во полето N: T=K\*N/2. За да се најде елемент треба да се направат просечно N/2 споредби. Може К/2 да се означи како константа па се добива Т=К\*N.

При бинарно пребарување времето е пропорционално на log(N): T=K\*log2(N).

Ознаката големо О не ја зема во предвид константата. Во практика при споредување на алгоритми не е важна брзината за специфичен микропроцесор или компајлер, туку само зависноста на Т од N. Затоа не е потребна константа. За линеарното пребарување времето на извршување е О(N), а за бинарното пребарување O(log N). Вметнување во неподредено поле се изведува за O(1) или константно време. Во табелата 2.4 дадени се времињата на извршување на операциите со полиња изразени со ознаката големо О.

Табела 2.4 Времиња на извршување на операции со полиња изразени со ознаката големо О

|  |  |
| --- | --- |
| алгоритам | Време на извршување изразено со ознаката големо О |
| Линеарно пребарување | O(N) |
| Бинарно пребарување | O(log N) |
| Вметнување во неподредено поле | O(1) |
| Вметнување во подредено поле | O(N) |
| Бришење од неподредено поле | O(N) |
| Бришење од подредено поле | O(N) |

На сликата 2.2 се претставени големо О релации меѓу времето и бројот на елементи. Време на извршување на алгоритам О(1) е одлично, O(log N ) е многу добро, O(N) е добро, а O(N2) е лошо.



Слика 2.2 Времиња на извршување изразени со ознаката големо О

Во неподредени полиња вметнувањето на елемент е брзо и изнесува О(1), но пребарувањето е бавно O(N). Кај подредените полиња пребарувањето е брзо O(log N) но вметнувањето е бавно O(n). Кај двата типа полиња, подредени и неподредени, бришењето изнесува O(N) бидејќи половина од елементите просечно треба да се преместат.

Понатаму ќе бидат опишани структури каде сите операции се извршуваат за време O(log N). Но за сметка на побрзото извршување овие структури се покомплексни. Друг недостаток на полињата е фиксната големина. Однапред треба да се предвиди големината на полето. Во другите поглавја ќе бидат опишани структури кои може да се прошируваат за време на извршување на програмата, поврзани листи.