
# Polja

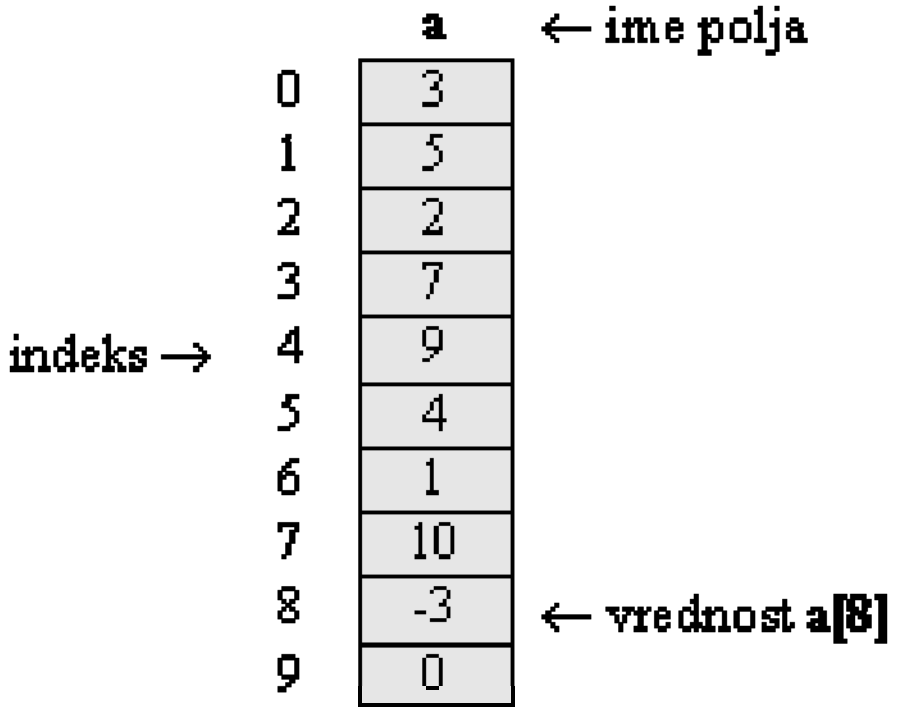
# Definicija

**Polje (engl. array) predstavlja skup elemenata iste vrste kojima se može direktno pristupati korišćenjem numeričkog indeksa.**

# Tipovi

- Jednodimenzionalna polja – **vektori**
- Dvodimenzionalna polja – **matrice**
- Višedimenzionalna polja

# Primer celobrojnog jednodim.polja



# Pristup elementima

$\alpha$ - adresu prvog elementa vektora A = ($a_\delta$, $\mathbf{a_{\delta+1}}$, **... $\mathbf{a_n}$**)
$\sigma$ - veličina elementa
$\delta$ - početnu vrednost indeksa

$$adresa(a[i]) = \alpha + \sigma \cdot (i - \delta)$$

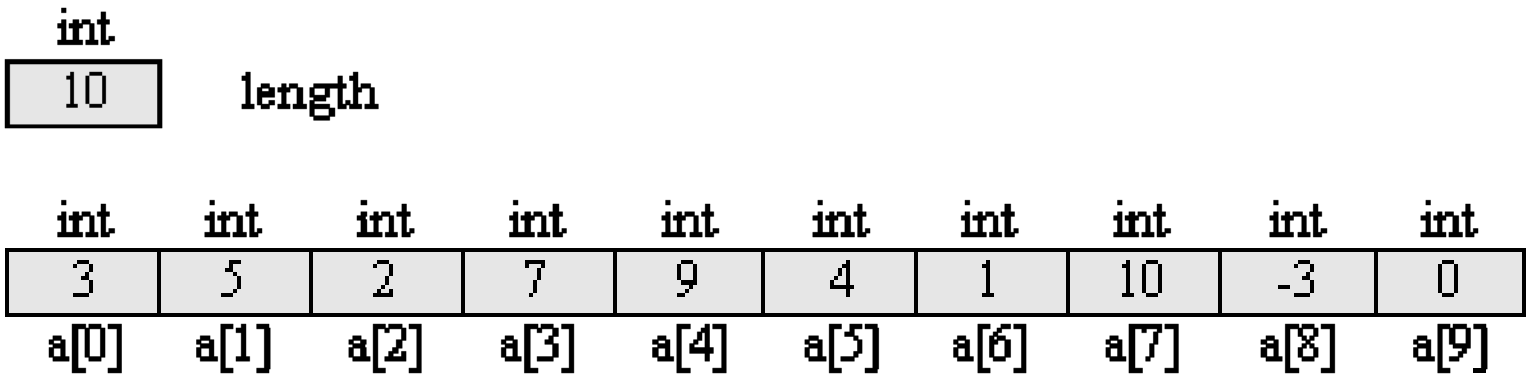
Kako je u C++ $\delta = 0$, dobija se:

$$adresa(a[i]) = \alpha + \sigma \cdot i$$

# Rad sa poljima u C/C++u

- Statička deklaracija
  - `int a[10];`
- Dinamička deklaracija
  - `int len = 50;`
  - `int* a = new int[len];`
- Dealokacija
  - `delete [] a;`
- Statička deklaracija sa dodelom vrednosti
  - `int a[10] = {1,2,3,4,5,6,7,8,9,0};`

# Memorijska reprezentacija
## jednodimenzionalnog polja



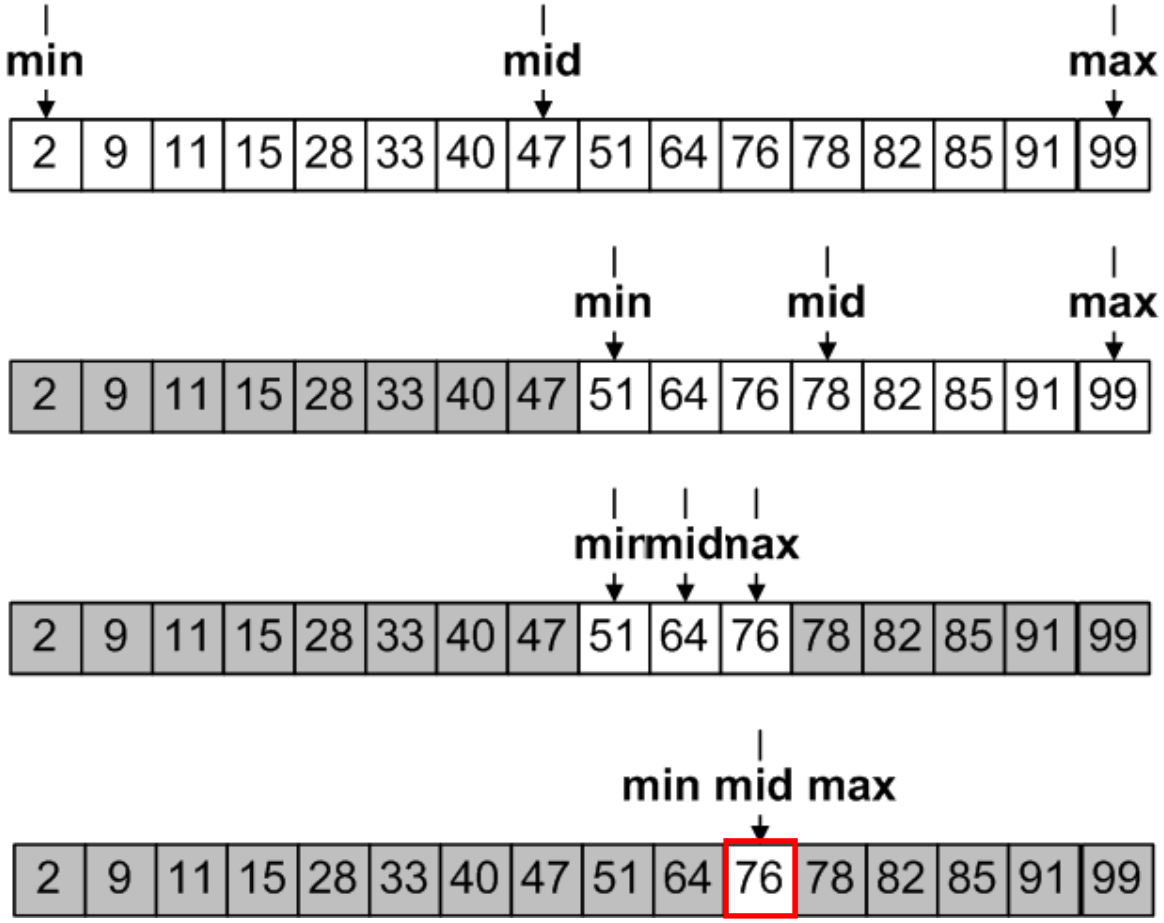
# Operacije

- **obilazak polja** (engl. *traversal*) - pristupa se svakom elementu polja i nad njim vrši neka obrada (na primer, provera vrednosti ili ažuriranje),
- **traženje** (engl. *search*) - traži se element zadate vrednosti,
- **umetanje** (engl. *insertion*) - dodaje se novi element polja na zadatu lokaciju,
- **brisanje** (engl. *deletion*) - iz polja se uklanja element zadate vrednosti ili sa zadate lokacije,
- **sortiranje** (engl. *sorting*) - elementi polja se reorganizuju prema nekoj vrsti uređenosti (kod numeričkih polja to je obično uređenje u rastući ili opadajući redosled vrednosti, kod alfanumeričkih polja to je alfabetska ili leksikografska uređenost ).

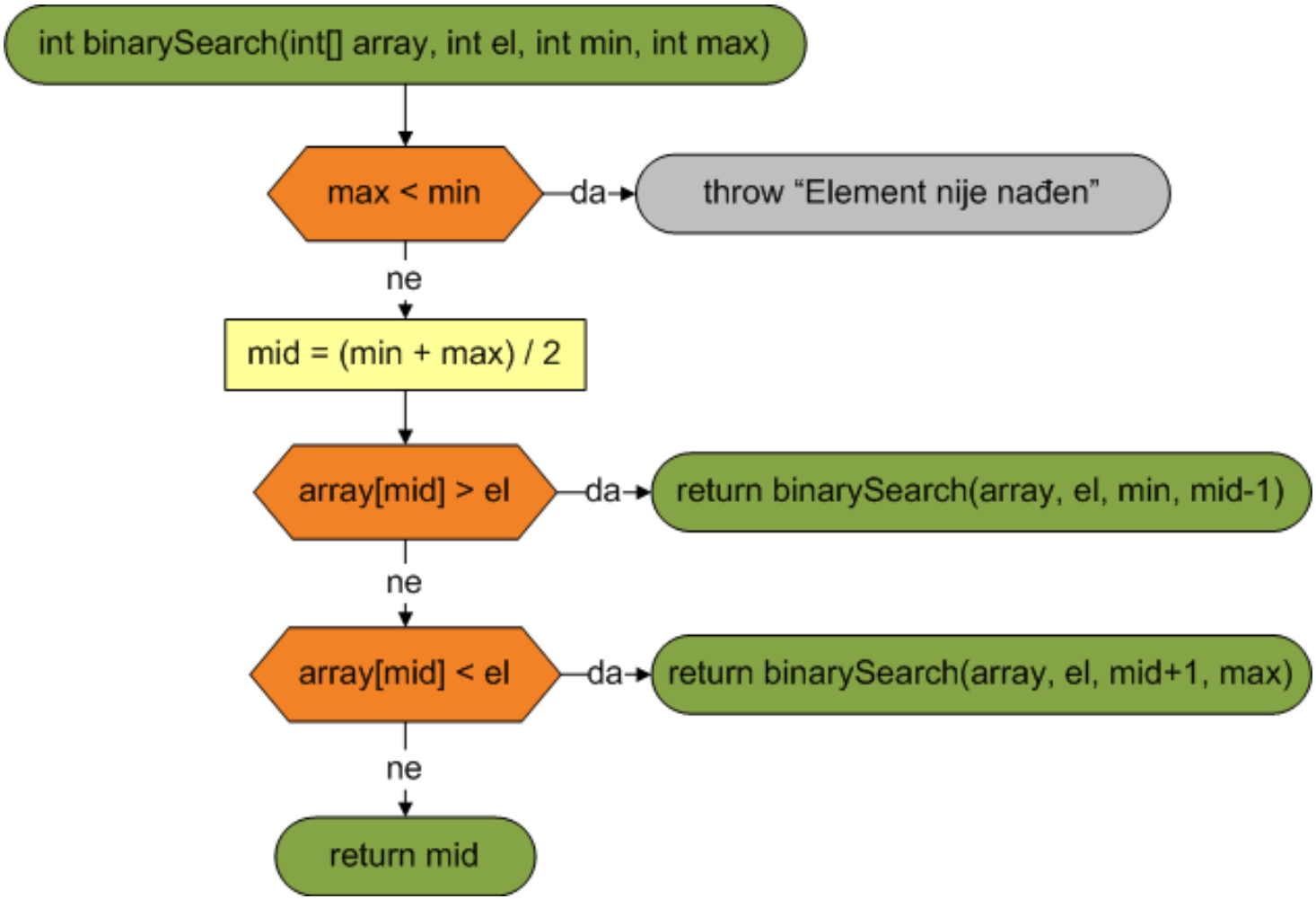
# Binarno traženje

**Da li se 76 nalazi u polju?**

76



# Binarno traženje

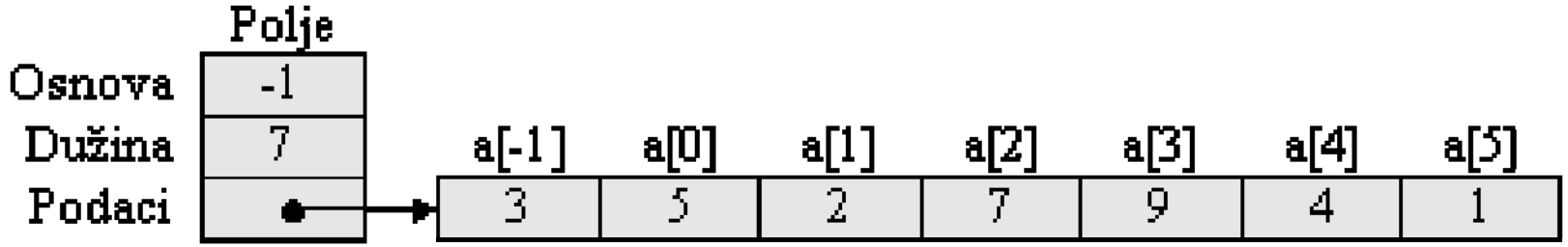


# Binarno traženje

```
int binarySearch(int[] array, int el, int min, int max)
{
// U vektoru array traži elemenat el u opsegu indeksa od min do max

        if (max < min)
            throw new SBPException("element nije nadjen");

        int mid = (min + max) / 2;
        if (array[mid] > el)
        {
                return binarySearch(array, el, min, mid-1);
        }
        else if (array[mid] < el)
        {
                return binarySearch(array, el, mid+1, max);
        }
        else
        {
                return mid;
        }
}
```

# Dinamička polja

Polja koja mogu menjati veličinu tokom vremena.



# 1D polja

```
template <class T>
class Array
{
protected:
        T* data;                    // vektor podataka
        long base;                  // osnova
        unsigned long length;       // duzina
public:
        Array(long m, long l){
                length = l;
                data = new T[l];
                base = m;
        };

        Array(long l) { length = l; data = new T[l]; base = 0L; };
        ~Array() { delete [] data; }
        void print();
```

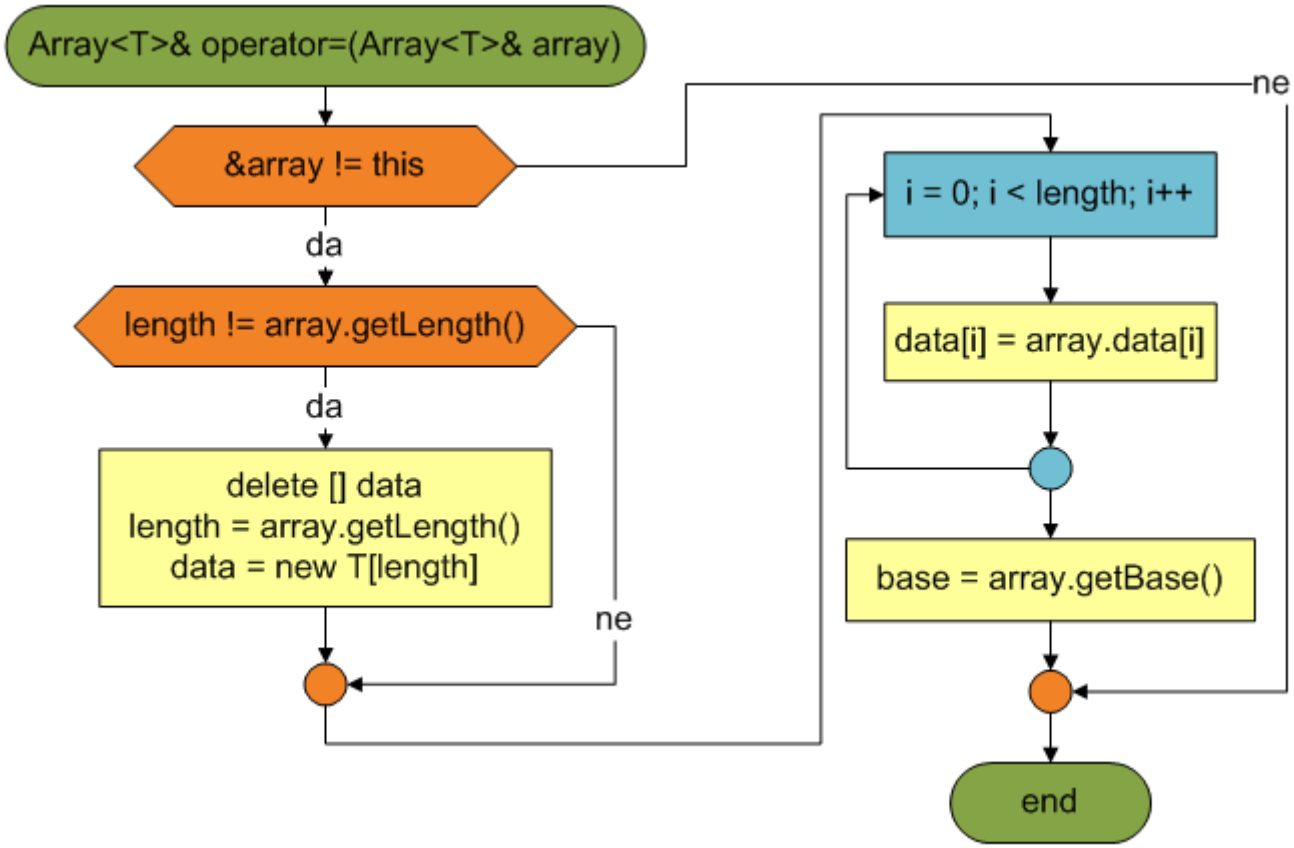
# 1D polja

```
// predefinisani operatori
        operator = (Array<T>& array);
        T& operator [] (long index);

        T* getData() { return data; }
        T& getAt(long index);

        long getBase() { return base; }
        unsigned long getLength() { return length; }

        inline void setBase (long NewBase) { base = NewBase; }
        void setLength (unsigned long NewLength);
        void setAt(T el, long index);
};
```

# 1D polja

```
template <class T>
void Array<T>::print()
{
        for(unsigned long i=0; i<length; i++)
                cout << "data["<<i<<"]=" << data[i] << endl;
}

template <class T>
Array<T>::operator = (Array<T>& array)
{
        if (&array != this)
        {
                if (length != array.getLength()){
                        if(data) delete [] data;
                        length = array.getLength();
                        data = new T[length];
                }
                for (unsigned long i = 0; i < length; i++)
                        data[i] = array.data[i];
                base = array.getBase();
        }
}
```

# Kopiranje sadržaja 1D polja



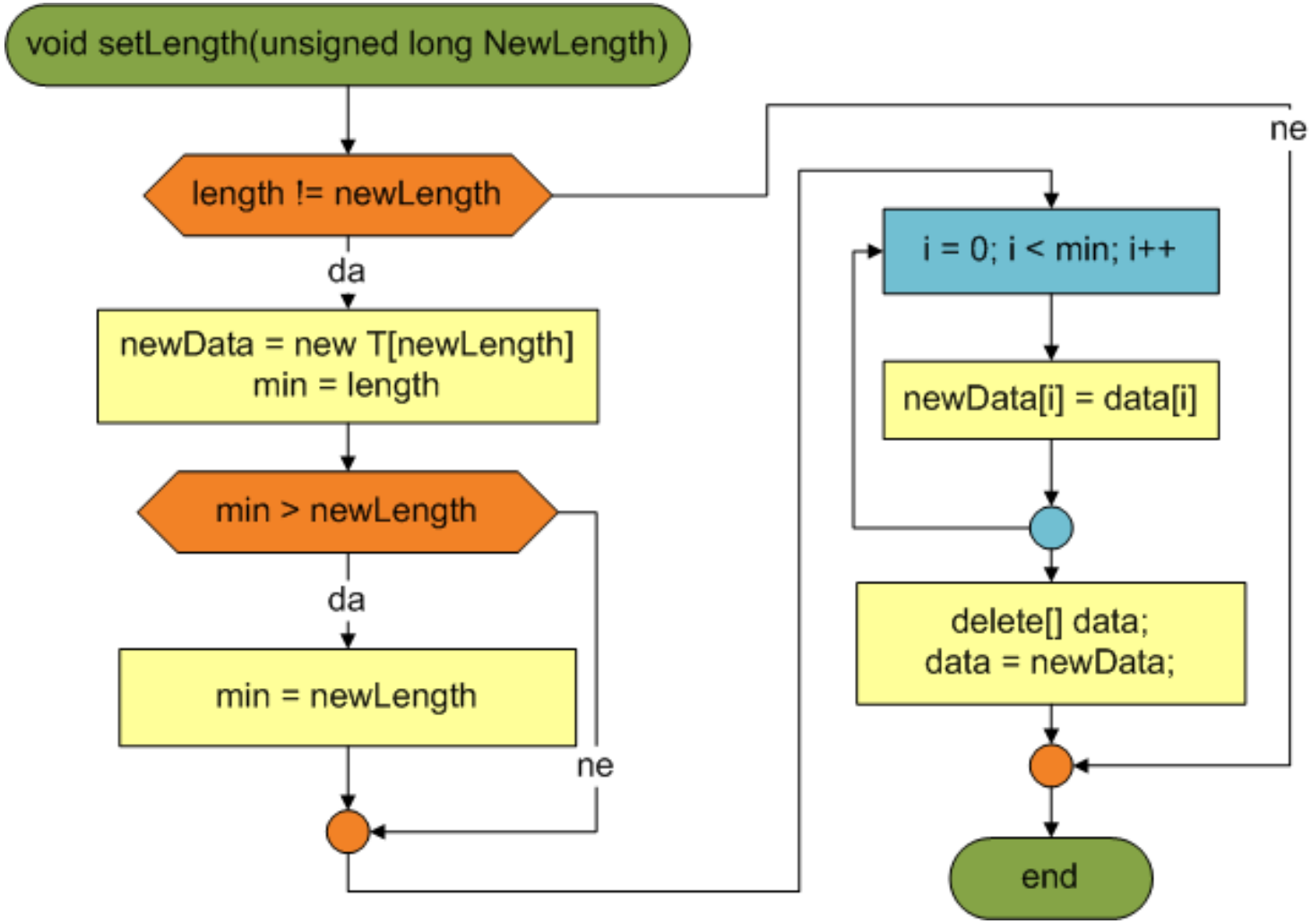
# 1D polja

```
template <class T>
T& Array<T>::operator [] (long index)
{
        unsigned long offset = index - base;
        if (offset >= length)
                throw new SBPException("Index out of bounds!");
        return data[offset];
}

template <class T>
T& Array<T>::getAt(long index)
{
        unsigned long offset = index - base;
        if (offset >= length)
                throw new SBPException("Index out of bounds!");
        return data[offset];
}
```

# 1D polja

```
template <class T>
void Array<T>::setAt(T el, long index)
{
        long offset = index - base;
        if (offset >= length)
                throw new SBPException("Index out of bounds!");
        data[offset]=el;
}

template <class T>
void Array<T>::setLength (unsigned long NewLength)
{
        if (length != newLength){
                T* newData = new T[newLength];
                int min = length;
                if( min > newLength ) min = newLength;
                for (int i = 0; i < min; i++)  newData[i] = data[i];
                delete [] data;
                data = newData;
        }
}
```

# Promena veličine polja



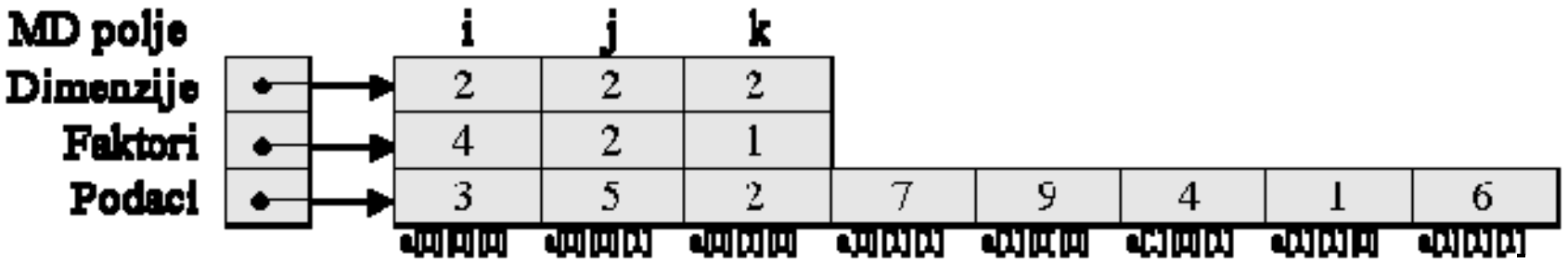
# Višedimenzionalna polja

Višedimenzionalno polje dimenzije N je kolekcija podataka kojima se pristupa preko N indesnih izraza (npr. A[i][j]).

$$adresa(a[i_0, i_1, \ldots i_{N-1}]) = \alpha + \sigma \cdot \sum_{k=0}^{N-1} faktori[k] \cdot i_k$$

$$faktori\,[k] = \begin{cases} 1 & k = N-1 \\ \prod_{j=k+1}^{N-1} dimenzije\,[j] & k < N-1 \end{cases}$$

# Memorijska reprezentacija višedimenzionalnog polja



# Zaglavlje klase višedimenzionalnog polja (MD)

```
template <class T>
class MultiDimensionalArray
{
protected:
        T* data;          // vektor podataka
        long* dim;        // dimenzije
        long* fact;       // faktori
        int dimensions; // dimenzionalnost (br.dim.)
public:
        MultiDimensionalArray (long* args, int n);
        inline T& getAt (long* indices, int n)
                { return data[getOffset (indices,n)]; }
        inline void setAt (T obj, long* indices, int n)
                {data[getOffset (indices,n)] = obj;}
protected:
        long getOffset (long* indices, int n);
};
```

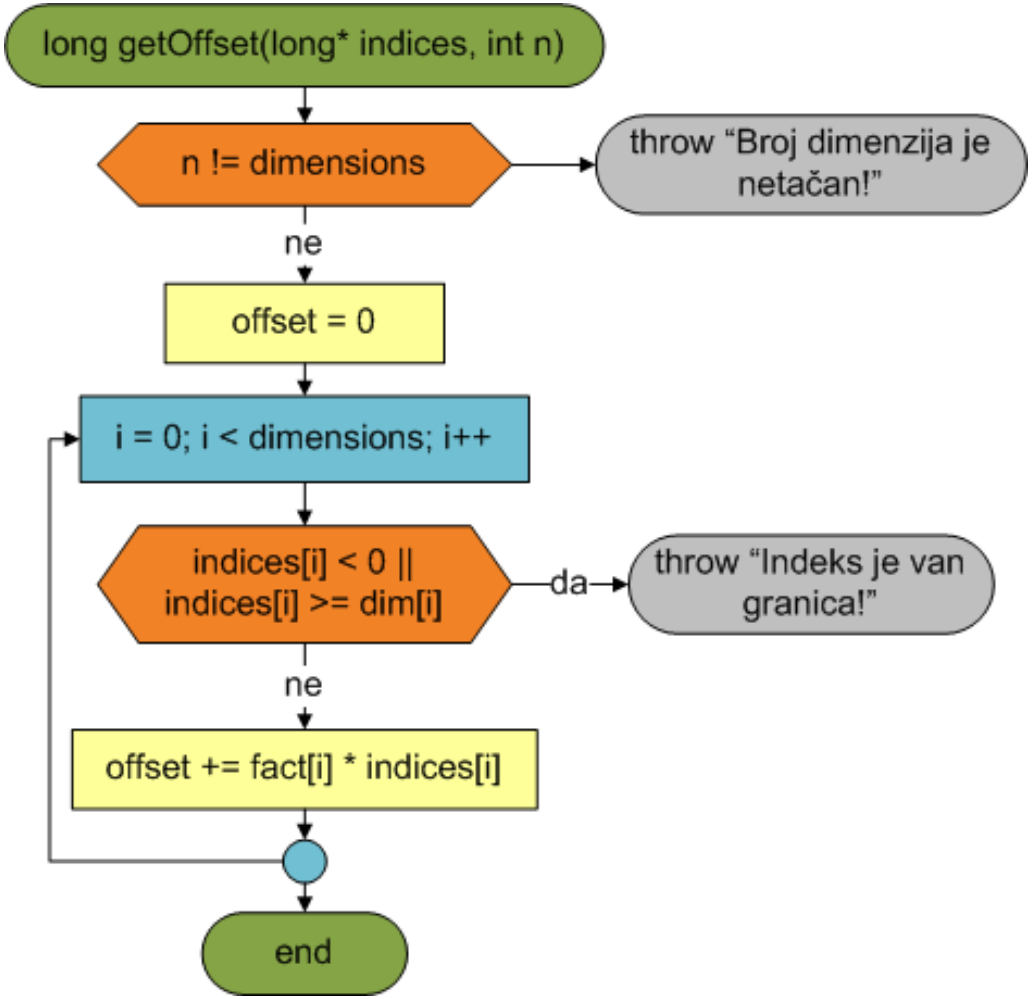
# Konstruktor MD polja

```
template <class T>
MultiDimensionalArray<T>::MultiDimensionalArray (long* arg, int n)
{
        dimensions = n;
        dim = new long[n];
        fact = new long[n];
        long product = 1;
        for (int i = n - 1; i >= 0; --i)
        {
                dim[i] = arg[i];
                fact[i] = product;
                product *= dim[i];
        }
        data = new T[product];
}
```

# Određivanje pomeraja

```
template <class T>
long MultiDimensionalArray<T>::getOffset (long* indices, int n)
{
        if (n != dimensions)
                throw new
                        SBPException("Broj dimenzija je netacan!");
        long offset = 0;
        for (int i = 0; i < dimensions; i++){
                if (indices[i] < 0 || indices[i] >= dim[i])
                        throw new SBPException("Indeks je van granica!");
                offset += fact[i] * indices[i];
        }
        return offset;
}
```

# Određivanje pomeraja



# Matrice

- dvodimenzionalna polja realnih brojeva
- prvi indeks – vrsta
- drugi indeks – kolona
- tipovi:
  - po obliku:
    - **kvadratne** (engl. *square matrix*)
    - **pravougaone** (engl. *rectangular matrix*)
  - po popunjenosti:
    - **retko posednuta** (engl. *sparse matrix*)
    - **gusta** (engl. *dense*)

# Specijalne matrice

$$\begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{bmatrix}$$

Gornje trougaona

$$\begin{bmatrix} x & 0 & 0 & 0 \\ x & x & 0 & 0 \\ x & x & x & 0 \\ x & x & x & x \end{bmatrix}$$

Donje trougaona

Max. br. elemenata: $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$

Matrica je simetrična ako je $a_{ij} = a_{ji}$ i tada je dovoljno memorisati samo elemente donje ili gornje trougaone matrice.

# Virtelna klasa matrice

```
class Matrix
{
public:
        virtual double get (int i, int j);          // Citanje vrednosti
        virtual void put (int i, int j, double d);  // Dodela vrednosti
        virtual Matrix& transpose ();           // Transponovanje matrice
        virtual Matrix& times (Matrix matrix);    // Mnozenje matrica
        virtual Matrix& plus (Matrix matrix);     // Sabiranje matrica
};
```

# Gusta matrica

```
class DenseMatrix: public Matrix
{
protected:
        long noRows;            // broj vrsta
        long noColumns;         // broj kolona
        double* data;           // polje sa podacima

public:
        DenseMatrix (long numOfRows, long numOfColumns){
                noRows = numOfRows;
                noColumns = numOfColumns;
                data = new double[ noRows * noColumns ];
        }
};
```

# Primer inicijalizacije

```
Array<int> pomArray(5);
pomArray[3]=2;
pomArray.print();
Array<int> pomArray2(6);
long ind[3]={2, 3, 4};
MultiDimensionalArray<int> mda((long*)ind, 3);
long ind2[3]={1, 2, 3};

try
{
        pomArray2[5] = 5;
        pomArray = pomArray2; // kopiranje elemenata polja
        pomArray.print();
        int pom = pomArray[5]; // preuzimanje 5-tog el. korišćenjem operatora []
        cout << pom << endl;
        pom = pomArray.getAt(5); // preuzimanje 5-tog el. korišćenjem funkcije
        cout << pom << endl;

        mda.setAt(12, (long*)ind2, 3); // mda[1][2][3] = 12
}
catch(SBPException* e)
{
        cout << e->GetError() << endl;
        delete e ;
}
```

# Primena polja - Sortiranje

- **Bubble** sort (metod mehurova)
- **Selection** sort (metod selekcije)
- **Insertion** sort (metod umetanja)
- **Merge** sort (metod mešanja)

# Bubble Sort

Upoređuju se dva po dva elementa i veći ide dalje, tako da u prvom prolazu najmanji element “isplivava” na prvu poziciju.

```
void bubblesort(int* data,int dim)
{
    int i, j;
    for( i = 0; i < dim-1; i++ )
      for( j = dim-1; j > i; j-- )
        if( data[j] < data[j-1] )
          swap(j, j-1);
}
```

Složenost: **$O(n^2)$**

# Selection Sort

Traži se indeks najmanjeg el. u celom vektoru i on menja mesto sa prvim el.
Nakon toga traži se najmanji u preostalom delu vektora i stavlja na 2. mesto itd.

```
void selectionsort(int* data,int dim)
{
  int i, j, least;
  for( i = 0; i < dim-1; i++ ){
    for( j=i+1,least=i; j<dim; j++ )
      if( data[j] < data[least] )
        least=j;
    swap(least,i);
  }
}
```

Složenost: $\mathbf{O(n^2)}$

# Insertion Sort

Polazeći od I-tog el. vektora, pomeraju se sve elementi levo od njega (koji su veći od I-tog) sve do el. manjeg od I-tog za po jedno mesto udesno, da pi se napravilo mesto za I-ti el.

```
void insertionsort(int* data,int dim)
{
  int i, j, tmp;
  for( i = 1; i < dim; i++ )
  {
    tmp = data[i];
    for( j = i;  j > 0 && tmp < data[j-1];  j-- )
      data[j] = data[j-1];
    data[j] = tmp;
  }
}
```

Složenost: **$O(n^2)$**

# Merge Sort

```
int* temp;
void mergesort(int*data, int dim)
{
  temp = new int[dim];
  mergesort(data, 0, dim-1);
  delete [] temp;
}
void mergesort(int*data, int first, int last)
{
  if( first < last ){
    int mid = ( first + last ) / 2;
    mergesort(data, first, mid);
    mergesort(data, mid+1, last);
    merge(data, first, last);
  }
}
```

# Merge Sort

```
void merge(int*data, int first, int last)
{
  int mid = ( first + last ) / 2;
  int i1 = 0, i2 = first, i3 = mid+1;
  while( i2 <= mid && i3 <= last )
    if( data[i2] < data[i3] )
      temp[i1++] = data[i2++];
    else
      temp[i1++] = data[i3++];
  while(i2 <= mid)
    temp[i1++] = data[i2++];
  while(i3 <= last)
    temp[i1++] = data[i3++];
  for( i1 = 0, i2 = first; i2 <= last; i1++, i2++ )
    data[i2] = temp[i1];
}
```

Složenost: $\mathbf{n \cdot \log_2 n}$

Može se ubrzati ako se rekurzija zameni iteracijom

Loša strana: zahteva dodatni mem. prostor (iste veličine kao i vektor koji se uređuje)

# Poređenje efikasnosti

| Algoritam | Najbolji slučaj | Najgori slučaj | Srednji slučaj |
|---|---|---|---|
| Bubble sort | $O(n^2)$ $n(n-1)/2$ | $O(n^2)$ $n(n-1)/2$ | $O(n^2)$ $n(n-1)/2$ |
| Selection sort | $O(n)$ $n-1$ | $O(n^2)$ $n(n-1)/2$ | $O(n^2)$ |
| Insertion sort | $O(n)$ $n-1$ | $O(n^2)$ $(n^2+3n-4)/2$ | $O(n^2)$ $(n^2+n-2)/4$ |
| Merge sort | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ |
| Quick sort | $n(\log_2 n+1)$ | $O(n^2)$ | $O(n \log_2 n)$ |
| Heap sort | $O(n)$ $(n \log_2 n)-O(n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ |

# Zadaci za proveru razumevanja gradiva

- Izračunati koliko operacija (množenja i sabiranja) se štedi kod pristupa M-dimenzionalnom polju korišćenjem unapred izračunatih faktora. (Uporediti broj operacija prilikom pristupa na "konvencionalan" način i sa unpred izračunatim faktorima.)
- Predložiti način za pamćenje trougaone matrice, tako da budu smešteni samo nenulti elementi.
  - Izvesti izraz za pristup elementima gornje trougaone matrice
  - Izvesti izraz za pristup elementima donje trougaone matrice
  - Projektovati klasu za rad sa donjom trougaonom matricom i implemetirati funkcije za upis i čitanje elementa (*double get(int i, int j)* i *void put(int i, int j, double value)*)
  - Izračunati koliko memorijskog prostora takav matrica zauzima u odnosu na standardnu "gustu" matricu, a koliko je pristup elementima složeniji