3

Nizovi

U ovom poglavlju se razmatraju neka od temeljnih svojstava nizova. Niz je jedna od najjednostavnijih, a istovremeno i najčešće korištenih struktura podataka u različitim područjima primjene, koja služi za čuvanje velikog broja istorodnih podataka. Isto tako, niz se često koristi i za implementaciju nekih drugih struktura podataka. U drugom dijelu se, kao primjer korištenja nizova, uvodi koncept liste kao strukture podataka, te se prikazuje sekvencijalni način implementacije liste upravo pomoću nizova. Dakako, nizovi se koriste i za implementaciju mnogih drugih struktura podataka, što će biti opisano kasnije u nekim drugim poglavljima.

3.1 Kontinualni prikaz nizova u linearnoj memoriji

Niz je linearno uređena struktura podataka koja se sastoji od konačnog broja homogenih elemenata. Pod homogenim elementima podrazumijevamo da su svi elementi istog tipa. Pod osobinom uređenosti podrazumijevamo da se tačno zna redoslijed elemenata u nizu. Najjednostavniji su jednodimenzionalni nizovi. Za specificiranje mjesta nekog elementa u nizu, koristimo indeks pozicije. Indeksi se uzimaju iz određenog raspona [$l..u$], gdje je

- l donja granica indeksa,
- u gornja granica indeksa.

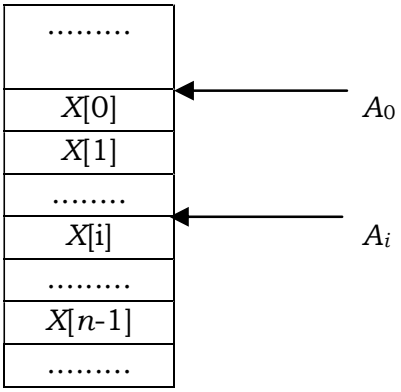
Donja granica l je obično 0 ili 1. Ukupan broj elemenata niza je u-l+1. Svakom elementu niza pristupamo navođenjem imena niza i indeksa elementa u zagradama. Na primjer, trećem elementu niza X pristupamo navođenjem X[2], pod pretpostavkom da je donja granica l=0. Za niz je važno poznavati njegove dimenzije da bismo mogli ispravno indeksirati, odnosno *pristupiti* njegovim elementima.

Za prikaz nizova u memoriji najčešće se koristi **sekvencijalni prikaz**, kojim se nizovi u memoriji implementiraju kao skup susjednih memorijskih lokacija. Na taj način se zapravo **logički koncept niza podudara** sa njegovom **fizičkom implementacijom**. Ovakav način organizacije podataka je neprikladan kada je potrebno umetati nove elemente na neku proizvoljnu poziciju ili kada je potrebno brisati element sa neke proizvoljne pozicije. Naime, prilikom umetanja novog elementa, potrebno je pomjerati sve elemente od dotične pozicije do kraja niza za jednu poziciju, da bi se oslobodio prostor za novi element.

S druge strane, da bi se sačuvao kontinuitet u organizaciji niza kao strukture, kod brisanja elementa iz niza je potrebno da se elementi iznad pozicije obrisanog elementa pomjere za jednu poziciju naniže. Prema tome, operacijama umetanja i brisanja se mjenjaju indeksi svih elemenata u nizu iznad pozicije na kojoj je izvršena operacija.

3.1.1 Alokacija jednodimenzionalnih nizova

Alokacija jednodimenzionalnih nizova je prikazana na slici 3.1. Ako pretpostavimo da je za smještanje jednog elementa niza potrebno s memorijskih riječi, onda je za smještanje čitavog jednodimenzionalnog niza, koji sadrži n elemenata, potrebno $n \cdot s$ memorijskih riječi.



Slika 3.1. Alokacija jednodimenzionalnog niza u memoriji

Ako početnu adresu prvog elementa u nizu označimo sa A_0, te ako pretpostavimo da za indeksiranje koristimo raspon indeksa $[0..n\text{-}1]$, onda se za pristup elementu $X[i]$ koristi tzv. *adresna funkcija*, koja nam omogućuje da na temelju početne adrese, indeksa elementa u nizu i veličine jednog elementa, izračunamo adresu A_i na kojoj je pohranjen element $X[i]$:

$$A_i = A_0 + i \cdot s,\ i = 0,\ 1,\ 2,\ \ldots,\ n\text{-}1. \tag{3.1}$$

Ključno svojstvo jednodimenzionalnog niza kao strukture podataka se ogleda u činjenici da se nekom elementu niza $X[i]$ pristupa na jedan efikasan način koristeći adresnu funkciju. Ovakav efikasan i jednostavan pristup elementima je zapravo posljedica toga što memoriju možemo promatrati kao jednodimenzionalnu strukturu sa linearno rastućim adresama, što omogućuje da su logički poredak elemenata niza i fizički poredak elemenata u memoriji isti.

Općenito, ako se umjesto raspona indeksa $0..n\text{-}1$, koristi raspon indeksa $l..u$, $u = l + n\text{-}1$, onda se adresa nekog elementa sa indeksom i, izračunava na sljedeći način:

$$A_i = A_l + (i\text{-}l) \cdot s,\ i = l,\ l+1,\ l+2,\ \ldots,\ l+n\text{-}1. \tag{3.2}$$

Izraz 3.2 je nešto složeniji od izraza 3.1, jer uključuje dodatno oduzimanje. Dakle, možemo izvesti zaključak da je izbor indeksa u rasponu $0...n\text{-}1$ sa stanovišta efikasnosti opravdan, pa neki programski jezici upravo uzimaju indeks 0 za indeks prvog elementa niza (C, C++,...).

3.1.2 Alokacija dvodimenzionalnih nizova

Da bi se u memoriji smjestili dvodimenzionalni nizovi potrebno je napraviti tzv. linearizaciju, pod kojom podrazumijevamo prevođenje dvodimenzionalnog poretka elemenata u jednodimenzionalni poredak. Linearizacija za niz $X[l_1..u_1,\ l_2..u_2]$ se treba napraviti tako da se dobije što jednostavnija adresna funkcija, koja će omogućiti jednostavno izračunavanje adrese $A_{i,j}$, za element niza $X[i, j]$. Dva najjednostavnija i najefikasnija pristupa za linearizaciju dvodimenzionalnog niza su *alokacija po redovima* i *alokacija po kolonama*.

Alokacija po redovima

Kod alokacije po redovima elementi dvodimenzionalnog niza se smještaju tako da se, počevši od početne adrese, prvo smješta prvi red, zatim drugi red, i tim redoslijedom, sve do zadnjeg reda (slika 3.2*b*). Uz pretpostavku da se koristi raspon indeksa i=0..m-1, j=0..n-1, adresna funkcija koja daje adresu nekog elementa $X[i, j]$ ima sljedeći oblik:

$$A_{i,j} = A_{0,0}+(i\cdot n+j)\cdot s,\ 0 \le i \le m\text{-}1,\ 0 \le j \le n\text{-}1, \tag{3.3}$$

gdje je:

- $A_{0,0}$ adresa prvog elementa u prvom redu i prvoj koloni;
- m, n broj redova i kolona, respektivno;
- s memorijska veličina jednog elementa.

Općenito, ako se za označavanje redova, umjesto raspona indeksa 0..m-1, koristi raspon indeksa $l_1 \ldots u_1$, te ako se za označavanje kolona, umjesto raspona indeksa 0..n-1, koristi raspon indeksa $l_2..u_2$, onda se adresa nekog elementa $X[i, j]$ izračunava na sljedeći način:

$$A_{i,j} = A_{l1,l2} +((i\text{-}l_1)\cdot n+j\text{-}\ l_2)\cdot s,\ l_1 \le i \le u_1,\ l_2 \le j \le u_2, \tag{3.4}$$

gdje je:

- $A_{l1,l2}$ adresa elementa niza u prvom redu i prvoj koloni;
- s broj memorijskih riječi za smještanje jednog elementa niza;
- n broj kolona u dvodimenzionalnom nizu, $n = u_2 - l_2+1$.

Primjetimo da u gornjem izrazu ne figurira broj m, koji predstavlja broj redova, $m = u_1 – l_1+1$. Alokacija po redovima je ilustrirana na slici 3.2*b*.

Alokacija po kolonama

Kod alokacije po kolonama elementi dvodimenzionalnog niza se smještaju tako da se prvo smjesti prva kolona, zatim druga kolona, i tim redoslijedom sve do zadnje kolone. Uz pretpostavku da se koristi raspon indeksa i = 0..m-1, j = 0..n-1, adresna funkcija ima sljedeći oblik:

$$A_{i,j} = A_{0,0}+(j\cdot m+i)\cdot s,\ 0 \le i \le m\text{-}1,\ 0 \le j \le n\text{-}1, \tag{3.5}$$

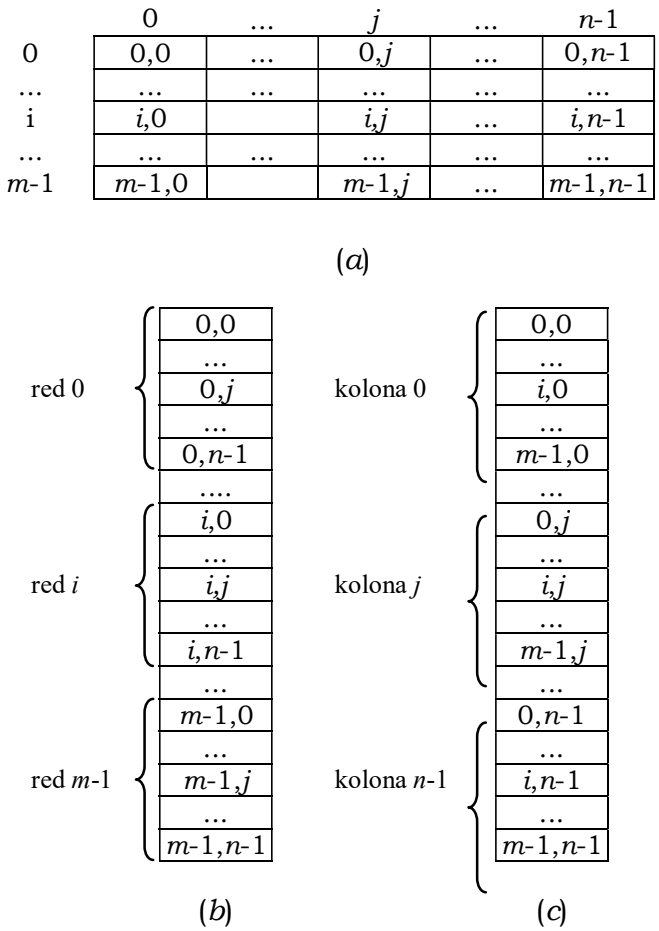
gdje oznake $A_{0,0}$, m, n i s imaju isto značenje kao i u izrazu 3.3.

Općenito, ako se za označavanje redova, umjesto raspona indeksa 0..m-1, koristi raspon indeksa $l_1..u_1$, te ako se za označavanje kolona, umjesto raspona indeksa 0..n-1,

koristi raspon indeksa $l_2..u_2$, onda se adresa nekog elementa $X[i, j]$ izračunava na sljedeći način:

$$A_{i,j} = A_{l1,l2} + (j-l_1)\cdot m + i-l_1)\cdot s,\ l_1 \le i \le u_1,\ l_2 \le j \le u_2, \tag{3.6}$$

gdje je oznake $A_{l1,l2}$, i s imaju isto značenje kao i u izrazu 3.4, dok m predstavlja broj redova, $m= u_1-l_1+1$. Primjetimo da u gornjem izrazu ne figurira broj n, koji predstavlja broj kolona, $n=u_2-l_2+1$. Alokacija po kolonama je ilustrirana na slici 3.2*c*.



Slika 3.2. Alokacija dvodimenzionalnog niza u memoriji: *a*) niz $X[0..m-1, 0..n-1]$, *b*) alokacija po redovima i *c*) alokacija po kolonama

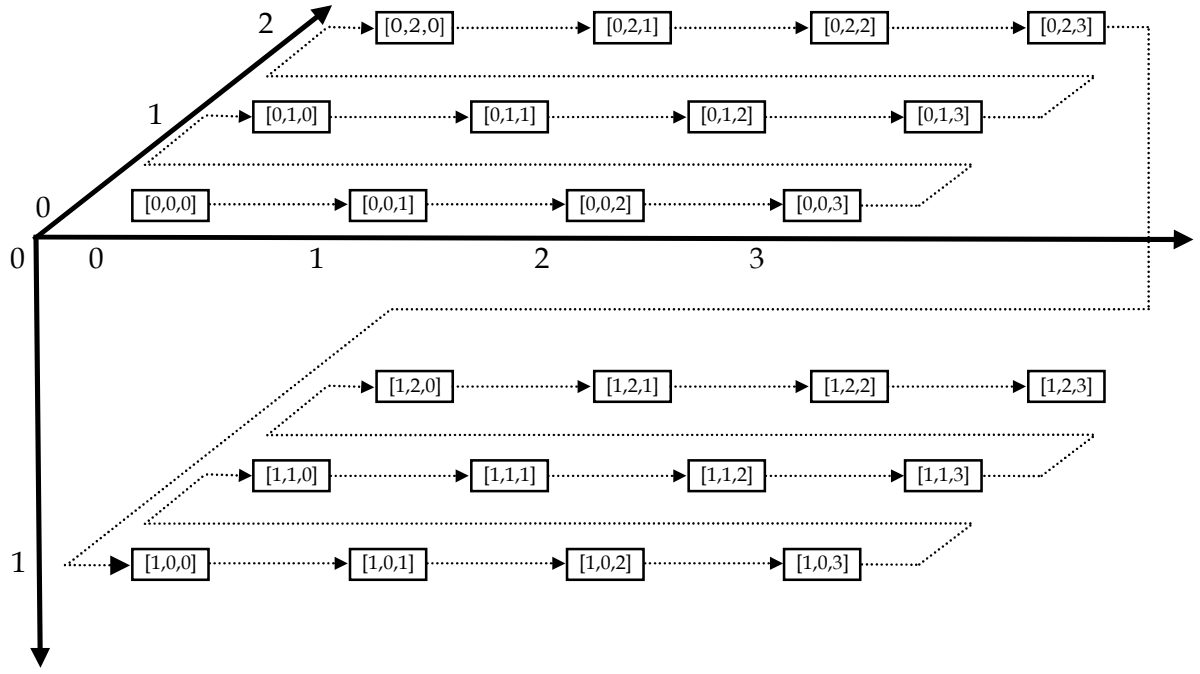
3.1.3 Alokacija višedimenzionalnih nizova

Na sličan način kao i kod dvodimenzionalnih nizova, da bi se u memoriji obavilo smještanje višedimenzionalnih nizova, potrebno je napraviti njihovu linearizaciju. Prethodno smo vidjeli da pojam smještanja po redovima možemo interpretirati kao smještanje kod kojeg prvo varira drugi indeks (broj kolone), a zatim prvi indeks (broj reda). Drugim riječima, prvo je varirao indeks najviše dimenzije niza, a zatim one manje dimenzije. Ako napravimo generalizaciju na n-dimenzionalni niz $X[0{:}m_1,0{:}m_2,\ldots,0{:}m_n]$, onda bi prema analogiji smještanja po redovima kod dvodimenzionalnog niza, mogli definirati smještanje elemenata sa sljedećim poretkom:

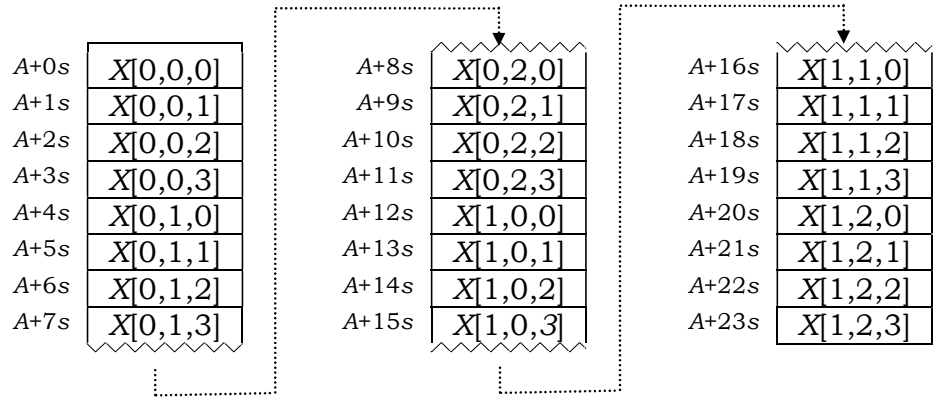
$X[0,\ldots,0,0]$
$X[0,\ldots,0,1]$
….
$X[0,\ldots 0,m_n]$
$X[0,\ldots 1,0]$
$X[0,\ldots 1,1]$
….
$X[0,\ldots 1,m_n]$
$X[0,\ldots 2,0]$
$X[0,\ldots 2,1]$
…..
…..
$X[m_1,m_2,\ldots m_n]$

Slika 3.3. Poredak elemenata n-dimenzionalnog niza

Za ilustraciju uzmimo primjer trodimenzionalnog niza $X[0..1,0..2,0..3]$. Prema tome, prva dimenzija ima veličinu 2 (m_1=1-0+1=2), druga dimenzija ima veličinu 3 (m_2=2-0+1) i treća dimenzija ima veličinu 4 (m_3=3-0+1). Na slici 3.4 je taj niz grafički ilustriran, pri čemu usmjerena linija koja povezuje elemente ilustrira princip na kojem će biti realiziran poredak elemenata u memoriji. Sekvencijalni poredak elemenata u memoriji je prikazan na slici 3.5.



Slika 3.4. Poredak elemenata niza $X[0:1,0:2,0:3]$



Slika 3.5. Sekvencijalni poredak elemenata u memoriji za niz X[0..1, 0..2, 0..3]

Da bismo krenuvši od prvog elementa $X[0,0,0]$, došli do elementa $X[i_1,i_2,i_3]$, gdje je npr. i_1=1, i_2=2 i i_3=2, prvo trebamo dostići podudaranje prvog indeksa (i_1=1), a to znači da prvo trebamo preći preko svih elemenata na prvoj ravni, odnosno trebamo proći grupu od 12 ($m_2 \cdot m_3$) elemenata da bismo došli do elementa $X[1,0,0]$. Zatim trebamo dostići podudaranje drugog indeksa (i_2=2), tako što ćemo preći preko dvije grupe (određeno indeksom i_2=2) od 4 elemenata (određeno veličinom dimenzije m_3) na drugoj ravni, da bismo došli do elementa $X[1,2,0]$. Na kraju je potrebno preći preko još 2 elementa (određeno indeksom i_3) da bismo došli do elementa $X[1,2,2]$. Prema tome, adresna funkcija, kojom izračunavamo adresu $A_{i1,i2,i3}$ nekog elementa $X[i_1,i_2,i_3]$ 3-dimenzionalnog niza sa veličinom dimenzija m_1, m_2 i m_3, ima sljedeći oblik:

$$A_{i1,i2,i3} = A_{0,0,0} + s(i_1 \cdot m_2 \cdot m_3) + i_2 \cdot m_3 + i_3, \tag{3.7}$$

gdje $A_{0,0,0}$ označava adresu prvog elementa niza $X[0,0,0]$. Odnosno, za primjer niza X[0..1,0..2,0..3] (gdje je m_1=2, m_2=3, m_3=4), adresna funkcija ima sljedeći oblik:

$$A_{i1,i2,i3} = A_{0,0,0} + s(i_1 \cdot 3 \cdot 4) + i_2 \cdot 4 + i_3). \tag{3.8}$$

Način smještanja po redovima možemo poopćiti i na n-dimenzionalne nizove. Ovdje pojam reda treba shvatiti u općenitijem smislu. Naravno, vizualizirano predočavanje n-dimenzionalnih nizova za n > 3 je otežano, ali ipak grafičku predodžbu za slučaj 3-dimenzionalnog niza prikazanu na slici 3.4, možemo iskoristiti da napravimo analogiju za razumijevanje smještanja općenito n-dimenzionalnih nizova. Označimo veličinu i-te dimenzije n-dimenzionalnog niza sa m_i, te koristimo za i-tu dimenziju raspon $0..m_i$-1.

Da bismo došli do nekog elementa $X[i_1, i_2,\ldots,i_n]$ ($0 \le i_j \le m_i$, j=1, 2, …, n) u n-dimenzionalnom nizu prvo trebamo postići podudaranje prvog indeksa i_1, odnosno trebamo doći do elementa $X[i_1,0,..0]$. Za ovakav skok je potrebno preći preko i_1 grupa elemenata, pri čemu svaka grupa ima $(m_2\text{-}1)(m_3\text{-}1)\ldots(m_n\text{-}1)$ elemenata. Nakon toga trebamo doći do elementa $X[i_1, i_2,\ldots0]$, da bismo postigli podudaranje drugog indeksa i_2. To ćemo postići tako što ćemo preći preko i_2 grupa, pri čemu svaka grupa ima $(m_3\text{-}1)(m_4\text{-}1)\ldots(m_n\text{-}1)$ elemenata. Ovakav način kretanja ćemo nastaviti i kroz ostale dimenzije, sve dok ne izjednačimo prvih n-1 indeksa, odnosno dok ne dođemo do elementa $X[i_1, i_2,\ldots i_{n-1}, 0]$. Na kraju je potrebno preći još preko i_n elemenata, da bismo stigli do elementa $X[i_1,i_2,\ldots i_n]$.

Prema tome, adresna funkcija, kojom izračunavamo adresu $A_{i1,...in}$ nekog elementa niza $X[i_1,i_2,\ldots,i_n]$, ima sljedeći oblik:

$$A_{i1,...in}=A_{0,...0}+s\cdot((i_1(m_2-1)(m_3-1)\ldots(m_n-1)+i_2(m_3-1)(m_4-1)\ldots(m_n-1)+\ldots+i_{n-1}(m_n-1)+i_n), \quad (3.9)$$

gdje je $A_{0,...,0}$ adresa prvog elementa niza $X[0,0,..0]$. Gornju adresnu funkciju možemo prikazati u sažetijem obliku:

$$A_{i1,...,in} = A_{0,...,0} + s\sum_{j=1}^{n} i_j c_j, \quad (3.10)$$

gdje je

$$c_j = \begin{cases} \prod_{k=j+1}^{n} (m_k - 1), za\ j = 1, 2, \ldots, n-1 \\ 1, za\ j = n \end{cases}$$

Na sličan način kao što smo izveli adresnu funkciju za smještanje *n*-dimenzionalnog niza *po redovima*, može se izvesti i adresna funkcija za smještanje *n*-dimenzionalnog niza *po kolonama*. Još jednom naglasimo da u slučaju *n*-dimenzionalnih nizova pojam reda i kolone treba shvatiti općenitije.

3.2 Implementacija liste pomoću nizova

Iako pod listama zapravo imamo neko intuitivno razumijevanje tog pojma, na ovom mjestu ćemo ipak uvesti neke pojmove. Listu možemo definirati kao uređenu sekvencu podatkovnih elemenata. Pod uređenom sekvencom zapravo podrazumijevamo da svaki element ima svoju poziciju. U okviru ove knjige će se podrazumijevati da su elementi liste istog tipa, iako konceptualno gledano to ne mora biti tako. Operacije koje su definirane nad listom, kao apstraktnim tipom podataka, ne ovise o tome kojeg su tipa elementi. Na primjer, lista, kao apstraktni tip podataka, može sadržavati cjelobrojne podatke, znakovne podatke, zapise o finansijskim transakcijama ili neke druge jednostavnije ili kompleksnije tipove podataka.

Za listu kažemo da je **prazna** kada ona ne sadrži niti jedan element. Broj elemenata koji je trenutno smješten u listi se naziva **dužina** liste. Obično se početak liste naziva **čelo** (eng. *head*), dok se kraj liste naziva **rep** (eng. *tail*) ili ponekada **začelje**. Čelo liste je uvijek na poziciji 0, dok je rep na poziciji zadnjeg elementa s indeksom koji je jednak veličini liste umanjenoj za 1. Formalno, listu koja sadži *n* elemenata možemo prikazati uređenom *n*-torkom:

$$(a_0,\ a_1,\ a_2,\ \ldots,\ a_{n-1}),$$

gdje a_i predstavlja elemente liste, $i = 0,1,2,\ldots n-1$.

Pri dizajniranju liste potrebno je razmotriti koje osnovne operacije lista treba podržavati i koje će atribute lista imati. Pošto listu na ovom mjestu implementiramo pomoću nizova, potrebno je u dizajnu liste *L* dodati atribut *L.kapacitet*, koji će predstavljati maksimalni mogući broj elemenata koje lista *L* može sadržavati. Osim toga, stvarnu veličinu liste ćemo prikazati atributom *L.velicina*, koji će predstavljati trenutni broj elemenata u listi. Lako zaključujemo da bi lista trebala moći rasti i smanjivati se **umetanjem**, odnosno **izbacivanjem** elemenata iz **tekuće pozicije**. Tekuću poziciju ćemo definirati dijeljenjem liste na **dvije particije**, koje su odvojene **zamišljenom pregradom**. Tekuću poziciju ćemo predstaviti atributom *L.tekuci*, koji će pokazivati indeks tekuće

pozicije. Za označavanje tekuće pozicije u **konfiguraciji liste** koristiti ćemo vertikalnu crtu, koja će prikazivati zamišljenu pregradu između dvije particije. Na primjer, konfiguraciju liste sa 5 cjelobrojnih podataka možemo prikazati na sljedeći način:

(15, 27 | 8, 43, 29),

pri čemu pregrada dijeli listu na lijevu particiju, koja ima dva elementa (15 i 27), i desnu particiju sa tri elementa (8, 43 i 29).

Nadalje, uvest ćemo i operaciju kojom će se na kraj liste moći direktno **dodati** novi element, pri čemu će tekuća pozicija ostati nepromijenjena. Također, uobičajeno je pristupati pojedinim elementima liste bilo samo radi čitanja, bilo radi eventualne promjene vrijednosti elementa ili pak izvođenja neke druge operacije, koja stvara učinak na tekućoj poziciji. Zbog toga ćemo uvesti operacije koje će **omogućiti pristup** sljedećem ili prethodnom elementu iz neke tekuće pozicije, zatim pristup čelu i začelju liste, kao i pristup poziciji s bilo kojim indeksom u dozvoljenom rasponu. I konačno, potrebno je da listu stvaramo i reinicijaliziramo.

Operacije kojima se tekuća pozicija postavlja na početak liste, kraj liste, prethodni element, sljedeći element ili na neku proizvoljnu poziciju *i*, su vrlo jednostavne i opisane su u tabelama 3.1-3.5, respektivno. Istaknimo jedino da procedura IDI-NA-PRETHODNI ne proizvodi nikakav učinak, ako je lijeva particija prazna, slično kao što ni procedura IDI-NA-SLJEDECI ne proizvodi nikakav učinak, ako je desna particija prazna. Sve ove operacije se izvode u vremenu $O(1)$.

Tabela 3.1

```
    IDI-NA-POCETAK (L)
1   L.tekuci = 0
```

Tabela 3.2

```
    IDI-NA-KRAJ (L)
1   L.tekuci =L.velicina
```

Tabela 3.3

```
    IDI-NA-PRETHODNI (L)
1   if (L.tekuci ≠ 0) then
2       L.tekuci = L.tekuci -1
3   end_if
```

Tabela 3.4

```
    IDI-NA-SLJEDECI (L)
1   if (L.tekuci < L.velicina) then
2       L.tekuci = L.tekuci+1
3   end_if
```

Tabela 3.5

```
    IDI-NA-POZICIJU (L, i)
1   if (i<0 or i >L.velicina) then
2       ERROR ("Indeks izvan dozvoljenog raspona")
3   end_if
4   L.tekuci = i
```

Na primjer, prethodno navedene procedure bi početnu konfiguraciju liste (15, 27 | 8, 43, 29, 12, 34) promijenile na sljedeći način:

IDI-NA-POCETAK (*L*)	→	(|15, 27, 8, 43, 29, 12, 34)
IDI-NA-KRAJ (*L*)	→	(15, 27, 8, 43, 29, 12, 34|)
IDI-NA-PRETHODNI (*L*)	→	(15|27, 8, 43, 29, 12, 34)
IDI-NA-SLJEDECI (*L*)	→	(15, 27, 8|43, 29, 12, 34)
IDI-NA-POZICIJU (*L*,5)	→	(15, 27, 8, 43, 29|12, 34)

Umetanje novog elementa u listu je opisano funkcijom UMETNI u tabeli 3.6. Ako u listi ima još slobodnih lokacija, funkcija pomjera sve elemente od tekuće pozicije do kraja liste za jedno mjesto udesno, a zatim tekućoj poziciji dodjeljuje novi element *x*. Dodatno se još za 1 povećava atribut *velicina*, te se vraća *true* kao znak uspješno obavljene operacije. Inače, ako u listi nema više slobodnih mjesta, funkcija vraća *false*, kao znak neuspješno obavljene operacije. Najbolji slučaj pri umetanju je kada se tekuća pozicija nalazi na kraju i tada se umetanje izvodi u vremenu $O(1)$. Najgori slučaj je u situaciji kada je tekuća pozicija na početku. Tada je potrebno pomjeriti svih n elemenata liste udesno, pa je vrijeme izvođenja $O(n)$. U prosječnom slučaju je potrebno pomjeriti $n/2$ elemenata, pa se i u prosječnom slučaju operacija izvodi u vremenu $O(n)$.

Tabela 3.6

```
    UMETNI (L, x)
1   if (L.velicina < L.kapacitet) then
2      for i = L.velicina downto L.tekuci + 1 do
3         L[i] = L[i-1]
4      end_for
5      L[L.tekuci] = x
6      L.velicina = L.velicina + 1
7      return true
8   else
9      return false
10  end_if
```

Na primjer, za sljedeću početnu konfiguraciju (15, 27|8, 43, 29), pozivanje funkcije UMETNI (*L*, 20) će promjeniti konfiguraciju liste u (15, 27|20, 8, 43, 29).

Operacija kojom se novi element dodaje na kraj liste je opisana funkcijom DODAJ u tabeli 3.7. Nakon utvrđivanja da u listi ima slobodnih pozicija, na poziciju s indeksom *L.velicina*, neposredno iza trenutno zadnjeg elementa, upisuje se novi element *x*, zatim se povećava atribut *velicina* za 1, te se vraća vrijednost *true*. Ako nema slobodnih pozicija funkcija vraća *false*. Operacija dodavanja se izvodi u konstantnom vremenu $O(1)$.

Tabela 3.7

```
    DODAJ (x)
1   if (L.velicina < L.kapacitet) then
2      L[L.velicina] = x
3      L.velicina = L.velicina+1
4      return true
5   else
6      return false
7   end_if
```

Na primjer, poziv funkcije DODAJ (*L*, 12) će sljedeću početnu konfiguraciju (15, 27 | 8, 43, 29) promjeniti u (15, 27|8, 43, 29, 12).

Operacija izbacivanja je opisana funkcijom IZBACI u tabeli 3.8. Ako je indeks tekuće pozicije pozicioniran tako da je desna particija prazna, funkcija signalizira grešku. U suprotnom, element na tekućoj poziciji se pamti u privremenom objektu *x*, te se svi elementi od tekuće pozicije do kraja liste pomjeraju za jedno mjesto ulijevo. Na kraju se atribut *velicina* umanjuje za 1, te funkcija vraća element koji se izbacuje. Najbolji slučaj pri izbacivanju je situacija kada se izbacuje zadnji element, u kojoj nema pomjeranja elemenata, pa se operacija izvodi u konstantnom vremenu $O(1)$. Najgori slučaj se pojavljuje kada se izbacuje prvi element. Tada je potrebno pomjeriti preostalih n-1 elemenata, te zaključujemo da se operacija izvodi u vremenu $O(n)$. U prosječnom slučaju je potrebno pomjeriti $(n-1)/2$ elemenata, pa se u prosjeku operacija izbacivanja izvodi u linearnom vremenu $O(n)$.

Tabela 3.8

```
    IZBACI (L)
1   if  (L.tekuci = L.velicina) then
2      ERROR ("Nista za izbaciti")
3   end_if
4   x = L[L.tekuci]
5   for i = L.tekuci to L.velicina-1 do
6      L[i] = L[i+1]
7   end_for
8   L.velicina = L.velicina -1
9   return x
```

Na primjer, poziv funkcije IZBACI (*L*) će početnu konfiguraciju liste (15, 27, 20, 8|43, 29) promjeniti u konfiguraciju (15, 27, 20, 8|29).

3.2.1 Primjer implementacije liste pomoću nizova u jeziku C++

U ovom primjeru je data implementacija liste pomoću nizova. Programski kod je organiziran u sljedeće datoteke:

- `lista.h`
- `nizlista.h`
- `nizlistatmp.h`
- `main.cpp`

U datoteci `lista.h` je definirana apstraktna generička klasa `Lista`. Ova klasa sadrži čiste virtualne metode, odnosno sadrži metode koje će trebati sadržavati bilo koja druga klasa koja će se derivirati iz nje. U ovom primjeru derivira se klasa `NizLista`, koja predstavlja implementaciju liste pomoću nizova. Isto tako, iz apstraktne klase `Lista` će se u kasnijim primjerima derivirati i neke druge klase pomoću kojih će se implementirati lista na neke druge načine. Sadržaj datoteke `lista.h` je:

lista.h

```
// Definicija apstraktne klase "Lista"
#ifndef LISTA_H
#define LISTA_H

template <typename InfoTip>
class Lista {
private:
  void operator =(const Lista&);      // Zaštita za dodjelu
  Lista(const Lista&);                // Zaštita za konstruktor kopije
public:
  Lista() {}                                  // Podrazumijevani konstruktor
  virtual ~Lista() {}                         // Bazni destruktor
  virtual void Brisi()= 0;
  virtual bool Umetni(const InfoTip&)= 0;   // Umetanje elementa
  virtual bool Dodaj(const InfoTip&)= 0;    // Dodavanje na kraj
  virtual InfoTip Izbaci() = 0;             // Izbacivanje tekućeg elementa
  virtual void IdiNaPocetak() = 0;          // Pozicioniranje na početak
  virtual void IdiNaKraj() = 0;             // Pozicioniranje na kraj
  virtual void IdiNaPrethodni() = 0;        // Pozicioniranje na prethodni
  virtual void IdiNaSljedeci() = 0;         // Pozicioniranje na sljedeći
  virtual void IdiNaPoziciju(int pos) =0;   // Pozicioniranje na i-tu poz.
  virtual int LDuzina() const = 0;          // Dužina lijeve particije
  virtual int DDuzina() const = 0;          // Dužina desne particije
  virtual const InfoTip& DajTekuciElement() const = 0; // Vraća tekući el.
};

#endif
```

Klasa `NizLista` je definirana u datoteci `nizlista.h`. Kao što je već rečeno, klasa `NizLista` je javno derivirana iz apstraktne klase `Lista`. Ova klasa sadrži atribute `kapacitet`, `duzina` i `tekuci`, te niz `L` u koji se smještaju elementi liste. Nadalje, klasa sadrži metode navedene u apstraktnoj klasi `Lista`, te dodatnu metodu `Prikazi`, koja služi za prikaz sadržaja liste. Sadržaj datoteke `nizlista.h` je sljedeći:

nizlista.h

```
// Definicija klase "NizLista"
#ifndef NIZLISTA_H
#define NIZLISTA_H

#include "lista.h"

template <typename InfoTip>
class NizLista : public Lista<InfoTip> {
protected:
  int kapacitet;                        // Kapacitet liste
  int duzina;                           // Dužina liste
  int tekuci;                           // Pozicija tekućeg elementa
  InfoTip* L;                           // Niz za elemente liste
public:
  NizLista(int size = 10);              // Konstruktor
  ~NizLista() { delete [] L; }          // Destruktor
  void Brisi();                         // Brisanje liste
  bool Umetni(const InfoTip& x);        // Umetanje
  bool Dodaj(const InfoTip& x);         // Dodavanje
  InfoTip Izbaci();                     // Izbacivanje
  void IdiNaPocetak() {tekuci = 0;}     // Pozicioniranje na početak
  void IdiNaKraj() {tekuci = duzina;}   // Pozicioniranje na kraj
  void IdiNaPrethodni() {               // Pozicioniranje na prethodni
    if (tekuci != 0)
      tekuci--;
  }
  void IdiNaSljedeci() {                // Pozicioniranje na sljedeći
    if (tekuci < duzina)
      tekuci++;
  }
  void IdiNaPoziciju(int i) {           // Pozicioniranje na i-tu poz.
    if ((i < 0) || (i > duzina))
      throw "Indeks izvan raspona";
    tekuci=i;
  }
  int LDuzina() const {return tekuci;}   // Dužina lijeve particije
  int DDuzina() const {return duzina - tekuci;}  // Dužina desne particije
  const InfoTip& DajTekuciElement() const {      // Vraća tekući el.
    if (DDuzina() <= 0) throw "Nema nista za vratiti!\n";
      return L[tekuci];
  }
  void Prikazi() const {                   // Prikazuje sadržaj liste
    for(int i=0; i < duzina; i++)
      cout << "Element " << i << ":" << L[i] << endl;
  }
};

#endif
```

U datoteci `nizlistatmp.h` su date implementacije metoda klase `NizLista`. Metode su dizajnirane u skladu sa prethodno datim opisom analognih procedura i funkcija u pseudokodu. Sadržaj datoteke `nizlistatmp.h` je:

nizlistatmp.h

```
// Definicija metoda klase NizLista
#ifndef NIZLISTATMP_H
#define NIZLISTATMP_H

#include "nizlista.h"

template<typename InfoTip>                      // Konstruktor

NizLista<InfoTip>::NizLista(int size = 10) {
  kapacitet = size;
  duzina = tekuci = 0;
  L = new InfoTip[kapacitet];
}

template<typename InfoTip>                      // Uništavanje
void NizLista<InfoTip>::Brisi() {
  delete [] L;
  duzina = tekuci = 0;
  L = new InfoTip[kapacitet];
}

template<typename InfoTip>                      // Umetanje
bool NizLista<InfoTip>::Umetni(const InfoTip& x) {
  if (duzina<kapacitet) {
    for(int i = duzina; i > tekuci; i--)
      L[i] = L[i-1];
    L[tekuci] = x;
    duzina++;
    return true;
  }
  else
    return false;
}

template<typename InfoTip>                      // Dodavanje na kraj
bool NizLista<InfoTip>::Dodaj(const InfoTip& x) {
  if (duzina < kapacitet) {
    L[duzina++] = x;
    return true;
  }
  else
    return false;
}

template<typename InfoTip>                      // Izbacivanje
InfoTip NizLista<InfoTip>::Izbaci() {
  if (DDuzina() <= 0)
    throw "Nista za brisati!\n";
  InfoTip x = L[tekuci];
  for (int i=tekuci; i<duzina-1; i++)
    L[i] = L[i+1];
  duzina--;
  return x;
}

#endif
```

U datoteci `main.cpp` se nalazi testni program. Sadržaj datoteke `main.cpp` je:

`main.cpp`

```
// Testni program za listu (sekvencijalnu i ulančanu implementaciju)

#include <iostream>
using namespace std;

#include "nizlistatmp.h"

int Menu() {                    // Ispis menu opcija
  int izbor;
  cout << endl <<"---Menu--------------------------------\n\n";
  cout << "1. Idi na poziciju\n";
  cout << "2. Idi na prethodni\n";
  cout << "3. Idi na sljedeci\n";
  cout << "4. Umetanje elementa na tekucu poziciju\n";
  cout << "5. Dodavanje elementa na kraj liste\n";
  cout << "6. Izbacivanje tekuceg elementa\n";
  cout << "7. Daj vrijednost tekuceg elementa\n";
  cout << "8. Prikazi sadrzaj liste\n";
  cout << "0. Izlaz\n\n";
  cout << endl << "------------------Izaberite opciju>" ;
  cin >> izbor;
  if (cin)
    return izbor;
  else {
    cin.clear();
    cin.ignore(100,'\n');
    return -1;
  }
}

int main() {
  NizLista<int> lista;
  int izbor, p, x;
  while (izbor = Menu()) {
    try {
      switch (izbor) {
        case 0:
          return 0;
        case 1:
          cout << "\nUnesite mjesto:";
          cin >> p;
          lista.IdiNaPoziciju(p);
          break;
        case 2:
          lista.IdiNaPrethodni();
          break;
        case 3:
          lista.IdiNaSljedeci();
          break;
        case 4:
          cout << "\nUnesite podatak:";
          cin >> x;
          if (lista.Umetni(x)) cout << "Cvor je umetnut"<<endl;
          else cout << "Cvor nije umetnut";
          break;
```

```
        case 5:
          cout << "\nUnesite podatak:";
          cin >> x;
          lista.Dodaj(x);
          cout << "\nCvor je dodan\n";
          break;
        case 6:
          lista.Izbaci();
          cout << "\nElement je izbacen\n";
          break;
        case 7:
          cout << "\nTekuci element:" << lista.DajTekuciElement() << endl;
          break;
        case 8:
          lista.Prikazi();
          break;
        default:
          cout << "Pogresan izbor\n";
      }
    }
    catch (const char greska[]) {
      cout << greska;
    }
    catch (...) {
      cout << "\nGreska u programu";
    }
  }
}
```

3.3 Pitanja i zadaci za vježbu

1. Ukratko obrazložite alokaciju jednodimenzionalnih nizova.
2. Navedite oblik adresne funkcije kod alokacije jednodimenzionalnih nizova, podrazumijevajući da se koristi raspon indeksa $l..u$.
3. Ukratko obrazložite alokaciju dvodimenzionalnih nizova po redovima.
4. Navedite oblik adresne funkcije kod alokacije dvodimenzionalnih nizova po redovima, podrazumijevajući da se koristi raspon indeksa $l_1..u_1$(prva dimenzija) i $l_2..u_2$ (druga dimenzija).
5. Ukratko obrazložite alokaciju dvodimenzionalnih nizova po kolonama.
6. Navedite oblik adresne funkcije kod alokacije dvodimenzionalnih nizova po kolonama, podrazumijevajući da se koristi raspon indeksa $l_1..u_1$ (prva dimenzija) i $l_2..u_2$ (druga dimenzija).
7. Ukratko obrazložite alokaciju višedimenzionalnih nizova „po redovima“.
8. Ukratko obrazložite alokaciju višedimenzionalnih nizova „po kolonama“.
9. Na sličan način kao što je u odjeljku 3.1.3 izvedena adresna funkciju za smještanje *n*-dimenzionalnog niza "*po redovima",* izvedite i adresnu funkcija za smještanje *n*-dimenzionalnog niza "*po kolonama"*. Dakako, pojam reda i kolone u slučaju *n*-dimenzionalnih nizova treba shvatiti općenitije, odnosno na sličan način kako je to prikazano u odjeljku 3.1.3.
10. U implementaciji liste pomoću nizova (sekvencijalna implementacija), koja je opisana u odjeljku 3.2, dodajte metodu `ObrniPoredak`. Metoda treba obrnuti poredak elemenata u listi, te izmjeniti vrijednost atributa *tekuci*, tako da lijeva i desna particija liste međusobno izmjene broj elemenata koje te particije sadrže.
11. Analizirajte i odredite efikasnost u *big-O* notaciji implementirane metode u zadatku 10.
12. U implementaciji liste pomoću nizova (sekvencijalna implementacija), koja je opisana u odjeljku 3.2, dodajte metode `NadjiMin` i `NadjiMax`, pomoću kojih se pronalaze minimalni, odnosno maksimalni element u listi. Obe metode trebaju vratiti indekse pozicija pronađenih elemenata, pri čemu, u slučaju višestrukog pojavljivanja tih elemenata na različitim pozicijama, treba vratiti manji indeks.
13. Analizirajte i odredite efikasnost u *big-O* notaciji implementiranih metoda u zadatku 12.
14. U implementaciji liste pomoću nizova (sekvencijalna implementacija), koja je opisana u odjeljku 3.2, dodajte metode `IzbaciMin` i `IzbaciMax`, pomoću kojih se iz liste izbacuju minimalni, odnosno maksimalni element. Obe metode trebaju vratiti pronađene elemente. U rješenju možete koristiti i metode implementirane u zadatku 12.
15. Analizirajte i odredite efikasnost u *big-O* notaciji implementiranih metoda u zadatku 14.

16. U implementaciji liste pomoću nizova (sekvencijalna implementacija), koja je opisana u odjeljku 3.2, dodajte metodu `IzbaciMdoN`, koja kao argumente uzima cijele brojeve *m* i *n*, te iz liste izbacuje sve elemente koji se nalaze na pozicijama koje su određene rasponom indeksa *m..n*.

17. Analizirajte i odredite efikasnost u *big-O* notaciji implementirane metode u zadatku 16.

18. U implementaciji liste pomoću nizova (sekvencijalna implementacija), koja je opisana u odjeljku 3.2, dodajte metodu `ZamjeniParticije`, koja sve elemente iz lijeve particije prebacuje u desnu particiju, dok elemente iz desne particije prebacuje u lijevu particiju. Na primjer, za početnu konfiguraciju liste (8, 5, 11, 3 | 12, 7, 1, 15, 6, 14, 2), metoda `ZamjeniParticije` treba kreirati sljedeću konfiguraciju liste (12, 7, 1, 15, 6, 14, 2 | 8, 5, 11, 3).

19. Analizirajte i odredite efikasnost u *big-O* notaciji implementirane metode u zadatku 18.

20. Implementacija liste pomoću nizova ima nedostatak predefiniranog kapaciteta. Jedno od mogućih rješenja problema prekoračenja kapaciteta bi moglo biti da se niz zamjeni novim većim nizom, kada god dođe do prekoračenja kapaciteta. Na primjer, trenutna veličina niza bi se mogla udvostručiti, kada dođe do navedenog prekoračenja. Napravite implementaciju liste pomoću nizova, koja bi koristila opisanu mogućnost zamjene niza sa nizom dvostruko većeg kapaciteta.