2.2. Paprasčiausios tiesinės duomenų struktūros

Šiame skyriuje susipažinsime su naudingomis duomenų struktūromis, kurias gauname iš vienakrypčio tiesinio sąrašo apribodami pastarojo veiksmų galimybes. Atkreipsime skaitytojų dėmesį į pastarąjį faktą: naujosios duomenų struktūros yra efektyvios ne todėl, kad išplečiame dinaminio sąrašo galimybes (vėliau įsitikinsime, kad ir šis kelias yra labai produktyvus), bet, atvirkčiai, pasinaudosime griežčiau apibrėžtomis jų veiksmų savybėmis.

2.2.1. Stekas

Stekas (angl. *stack*) yra labai dažnai naudojama duomenų struktūra. Ją apibūdina principas "paskutinis įeina, pirmasis išeina" (angl. *LIFO - Last In, First Out*). Elementai įrašomi ir šalinami iš sąrašo pradžios.

Tokią taisyklę naudojame ir buityje, pavyzdžiui pažvelkime į padėklų krūvą valgykloje: padėklas, kuris paskutinis padedamas ant viršaus, bus paimtas pirmas, o apatinis padėklas, kuris buvo padėtas pirmas, bus paimtas paskutinis.

Kaip ir kitoms duomenų struktūroms turime apibrėžti tokius steko veiksmus:

- sukurti tuščią steką;
- įterpti elementą į steką;
- patikrinti, ar stekas yra tuščias;
- perskaityti viršutinio elemento reikšmę;
- išimti viršutinį elementą iš steko;

Priklausomai nuo steko realizavimo būdo, prieš įterpiant naują elementą, gali tekti patikrinti ar stekas nėra pilnas, o prieš išimant elementą iš steko tenka patikrinti ar jis nėra tuščias.

Nagrinėsime du steko realizavimo būdus.

Steko realizacija masyve

Steką apibrėžiame panaudodami įrašo duomenų struktūrą: jį sudaro du laukai – elementų masyvas ir viršūnės rodyklė.

```
struct stack{
  T data[N];
  int top=0;
}
```

Kadangi masyvo dydis yra fiksuotas, tai tokiame steke galėsime saugoti ne daugiau nei N elementų. Tačiau tokia situacija yra būdinga daugeliui taikomųjų uždavinių, aprašomų steko duomenų struktūra. Pavyzdžiui padėklai saugomi stove, kuriame galime padėti tik numatytą jų skaičių, į pistoleto apkabą taip pat galime įdėti tik fiskuotą skaičių šovinių.

Tuščiame steke viršūnės rodyklė *top* yra lygi nuliui. Stekas yra pilnas, jei $top = N$.

Elementas įterpiamas į steką tokia procedūra

```
push(T e){
  if (top < N){
    data[top] = e;
    top += 1;
  }
}
```

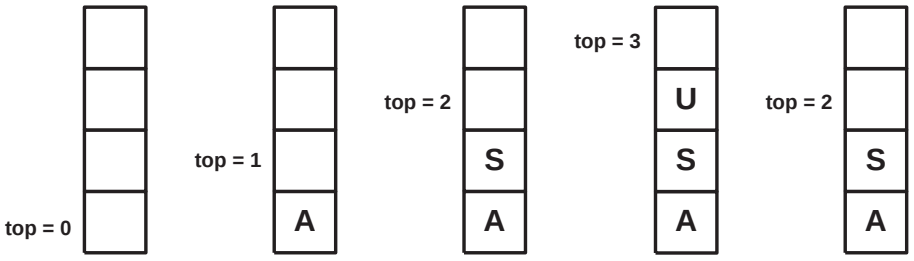
Šioje realizacijoje nieko nedarome, jei stekas jau pilnas. Todėl vartotojas turi pats pasirūpinti steko pilnumo sąlygos tikrinimu

```
int stackFull(){
  int full = 0;
  if (top ==  N)
    full = 1;
return (full);
}
```

2.8 paveiksle pavaizduotas stekas, į kurį įterpiamos raidės A, S, U, o po to raidė U yra pašalinama.

Elementas išimamas iš steko `pop`() procedūros pagalba

```
T pop(){
  if (top > 0){
    top -= 1;
    return (data[top]);
  }
}
```



2.8 pav. Steko realizavimas masyve: a) tuščias masyvas, b) push(A), c) push(S), d) push(U), e) pop().

Ir šioje procdūroje nedarome nieko, jei stekas yra tuščias. Taigi vartotojas pats turi patikrinti ar stekas yra tuščias, tai atlieka procedūra

```
int stackEmpty(){
  int empty = 0;
  if (top ==  0)
    empty = 1;
return (empty);
}
```

Procedūra readTop tik perskaito viršutinio elemento reikšmę, bet jo neišima iš steko:

```
T readTop(){
  if (top > 0)
    return (data[top-1]);
}
```

Steko realizacija dinaminiame sąraše

Naudodami tiesinį dinaminį sąrašą galime saugoti bet kokio ilgio steką, visada bus išskirta tik tiek kompiuterio atminties, kiek yra elementų. Įterpimo į steką veiksmas push vykdomas tiesinio sąrašo procedūra insertFront, o elemento šalinimas iš steko pop atliekmas panaudojant deleteFront veiksmą. Kaip pavyzdį pateiksime push procedūrą:

```
push(T e) {
  node *v;
  v = new node;
  (*v).T = e;
  if ( listStart == NULL) // Pirmas elementas
    (*v).next = NULL ;
  else
    (*v).next = (*listStart).next;
  listStart = v;
}
```

Kaip matome steko duomenų struktūra gali būti realizuota keliais būdais. Tokioje situacijoje labai patogu naudoti objektinio programavimo technologiją, pvz. sukurti C++ specialią steko klasę. Šios klasės vartotojas gali ir nežinoti, kaip realizuotas pats stekas, o jo programos veikimo teisingumas (bet ne efektyvumas) nepriklauso nuo steko realizavimo būdo.

2.2.2. Steko taikymai

Stekai labai plačiai naudojami įvairių algoritmų realizacijose, informatikoje, kompiuterių programų vykdyme. Pavyzdžiui iš programos teksto sudaromas vykdomųjų komandų stekas.

Šiame paragrafe susipažinsime su trijų uždavinių sprendimu: rekursijos algoritmų realizavimu, aritmetinių išraiškų atvaizdavimu *postfix* forma ir šia forma užrašytų aritmetinių išraiškų skaitinės reikšmės skaičiavimu.

Aritmetinės išraiškos vaizdavimas postfix forma

Dviejų skaičių a ir b sumą dažniausiai žymime $a + b$, toks užrašas yra vadinamas aritmetinės išraiškos *infix* forma. Čia aritmetinis veiksmas yra užrašomas tarp dviejų operandų.

Kalkuliatoriuose aritmetinės išraiškos užrašomos *prefix* forma, kai iš pradžių rašomas aritmetinis veiksmas, o po to operandai. Sumos prefix forma yra $+ab$.

Šiuolaikiniuose kompiuteriuose aritmetinės išraiškos vaizduojamos *postfix* forma, kai iš pradžių rašomi operandai, o po to aritmetinis veiksmas. Sumos postfix forma yra $ab+$.

Prefix ir postfix formos yra labai patogios todėl, kad nereikia naudoti skliaustų, norint nurodyti veiksmų atlikimo eiliškumą.

2.2 pavyzdys. Aritmetinių išraiškų postfix formos analizė. Nagrinėkime aritmetinę išraišką $a+b*c$. Užrašysime jos postfix formą

$$\begin{aligned} a+b*c &\longrightarrow a+(b*c) \longrightarrow a+(b\,c*) \\ &\longrightarrow a\,(b\,c*)+ \longrightarrow a\,b\,c\,*\,+\,. \end{aligned}$$

Kitos aritmetinės išraiškos $(a+b)*c$ postfix forma yra skirtinga ir jai užrašyti nereikia skliaustų, kurie buvo būtini infix formos išraiškoje

$$(a+b)*c \longrightarrow (a\,b+)*c \longrightarrow (a\,b+)\,c* \longrightarrow a\,b\,+\,c\,*\,.$$

Priminsime aritmetinių veiksmų prioritetus:

- Aukščiausią prioritetą turi kėlimo laipsniu veiksmas, kurį žymėsime simboliu $\wedge$:
$$a^b = a \wedge b\,.$$
Šis veiksmas yra atliekamas iš dešinės į kairę, t.y. naujasis laipsnio kėlimas yra aukštesnio prioriteto veiksmas
$$a \wedge b \wedge c = a \wedge (b \wedge c)\,.$$

- Žemesnį prioritetą turi daugybos ir dalybos veiksmai. Jų eiliškumas yra iš kairės į dešinę
$$a * b/c = (a * b)/c\,.$$

- Sumavimo ir atimties veiksmų prioritetas yra žemiausias, jų eiliškumas irgi iš kairės į dešinę
$$a - b + c = (a - b) + c\,.$$

- Skliaustuose esantis reiškinys interpretuojamas kaip vienas operandas ir apskaičiuojamas prieš kitus veiksmus (jo prioritetas yra aukštesnis už bet kurią aritmetinę operaciją).

Dabar pateiksime algoritmą, kuriuo *infix* formos išraišką konvertuojame į *postfix* formą. Jį realizuodami esminiai remsimės *steko* savybėmis.

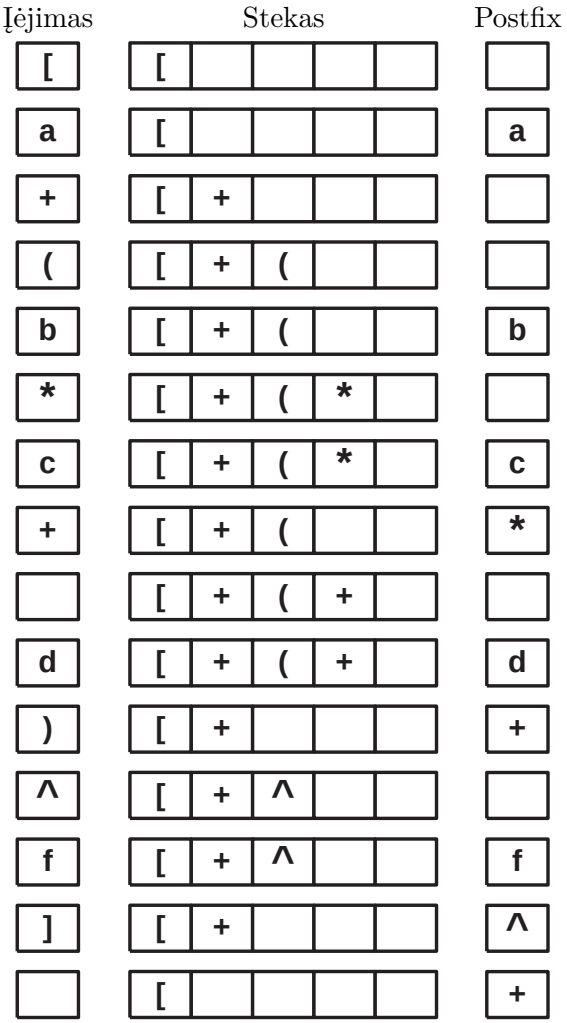
Infix išraiškos užrašymas postfix forma

```
(1)  Infix išraišką papildome pradžios " [ " ir pabaigos " ] " simboliais.
(2)  s = readDataFile();
(3)  select case (s):
     (4)  case ( "[",  "(" )
               (5)  push(s);
     (6)  case ( s  yra operandas  )
               (7)  print(s);
     (8)  case ( ∧, *,  /,  +,  − )
               (9)  h = readTop();
               (10) select case (h):
                    (11) case ( "[",  "(",  h ≺ s )
                              (12) push(s);
                    (12) case ( s ≺= h )
                              (13) h = pop();
                              (14) print(h);
                              (15) goto 9;
     (16) case ( ")" )
               (17) h = pop();
               (18) while ( h != "(" )
                         (19) print(h);
                         (20) h = pop();
     (21) case ( "]" )
               (22) h = pop();
               (23) while ( h != "[" )
                         (24) print(h);
                         (25) h = pop();
(26) if (s != "]") goto 2;
```

2.3 pavyzdys. Infix aritmetinės išraiškos užrašymas postfix forma. Nagrinėkime *infix* forma užrašytą aritmetinę išraišką $a + (b * c + d) \wedge f$. Panaudodami pateiktąjį algoritmą užrašysime ją *postfix* forma. Iš pradžių šią išraišką papildome pradžios ir pabaigos požymiais $[a + (b * c + d) \wedge f]$. Tolimesnė skaičiavimo eiga pateikta 2.9 paveiksle.

Taigi aritmetinės išraiškos $a + (b * c + d) \wedge f$ postfix forma yra

$$abc * d + f \wedge + .$$



2.9 pav. Aritmetinės išraiškos konvertavimas į postfix formą.

Postfix formos išraiškos skaitinės reikšmės skaičiavimas

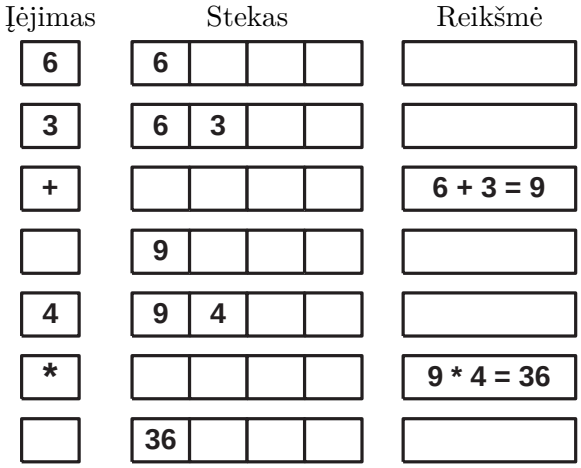
Dabar sudarysime algoritmą, kuriuo apskaičiuojame aritmetinės išraiškos, užrašytos *postfix* forma, reikšmę. Vėl naudosime steko duomenų struktūrą.

Pagrindinis algoritmo sudarymo principas yra paprastas: naujos aritmetinės operacijos operandai yra prieš tai apskaičiuotos dviejų aritmetinių veiksmų reikšmės.

Postfix išraiškos reikšmės skaičiavimas

```
(1) s = readDataFile();
(2) if (s yra skaičius)
        (3) push(s);
(4) else
        (5) b = pop();
        (6) a = pop();
        (7) c = a op(s) b;
        (8) push(c);
(9) if (yra duomenų) goto 1;
```

2.4 pavyzdys. Aritmetinės išraiškos reikšmės skaičiavimas. Raskime *postfix* išraiškos $6\ 3 + 4*$, kuri apibrėžia aritmetinę išraišką $(6 + 3) * 4$, reikšmę. Skaičiavimo eiga pateikta 2.10 paveiksle.



2.10 pav. Aritmetinės išraiškos $6\ 3\ +\ 4*$ reikšmės skaičiavimas.

2.2.3. Eilė

Kita dažnai naudojama duomenų struktūra yra *eilė* (angl. *queue*). Ją apibūdina principas "kas pirmas įeina, tas pirmas ir išeina" (angl. *FIFO - First In, First Out*). Elementai įrašomi į sąrašo pradžią, o šalinami iš jo galo.

Eilės taisyklę naudojame ir buityje: taip aptarnaujami klientai daugumoje parduotuvių, įstaigų. Kai kada eilės yra prioritetinės, pavyzdžiui poliklinikoje pirmiausia aptarnaujami karščiuojantys ligoniai, po to ligoniai, turintys lengvatų, ir tik vėliau visi likę pacientai. Kiekvienos grupės ligoniai vėl sudaro atskiras eiles.

Apibrėžiame tokius eilės veiksmus:

- sukurti tuščią eilę;
- įterpti elementą į eilę;
- patikrinti, ar eilė yra tuščia;
- perskaityti pirmojo elemento reikšmę;
- išimti pirmąjį elementą iš eilės;

Priklausomai nuo eilės realizavimo būdo, prieš įterpiant naują elementą, gali tekti patikrinti ar ji nėra pilna.

Eilės realizacija masyve

Eilę apibrėžiame panaudodami įrašo duomenų struktūrą: ją sudaro trys laukai – elementų masyvas ir eilės pradžios bei pabaigos rodyklės.

```
struct queue{
  T data[N];
  int start = -1;
  int end = -1;
}
```

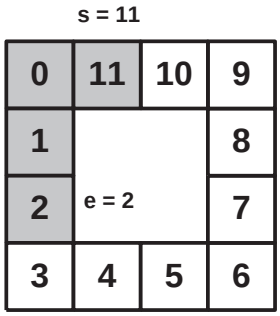
Pradinės rodyklių reikšmės ir apibrėžia tuščią eilę.

Kadangi masyvo dydis yra fiksuotas, tai tokioje eilėje galėsime saugoti ne daugiau nei N elementų. Elementai įterpiami į eilės galą, o šalinami iš eilės pradžios, taigi eilė "juda" masyve, todėl laikysime, kad masyvas yra uždaras ciklas, t.y. po data[N-1] vėl eina data[0] elementas. Todėl naudinga apibrėžti metodus:

```
int nextIndex(int i){
  int j = i+1;
  if (j ==  N)
    j = 0;
return (j);
}
```

```
int previousIndex(int i){
  int j = i-1;
  if (j == -1 )
    j = N-1;
return (j);
}
```

Eilės pavyzdys yra pateiktas 2.11 paveiksle.



2.11 pav. Eilės realizavimas masyve: eilės pradžia start=11, eilės pabaiga end=2.

Naujo elemento įrašymo į eilę veiksmą žymėsime put(T e). Prieš vykdant šią operaciją gali tekti patikrinti ar eilė jau pilna:

```
int queueFull(){
  int full = 0;
  if (nextIndex(end) == start)
    full = 1;
return (full);
}

put(T e){
  if ( queueEmpty() ){ // Tuščia eilė
    data[0] = e;
    start = 0;  end = 0;
  }else{
    end = nextIndex(end);
    data[end] = e;
  }
}
```

Elemento išėmimo iš eilės veiksmą žymėsime get(). Prieš vykdant šią operaciją gali tekti patikrinti ar eilė yra tuščia:

```
int queueEmpty(){
  int empty = 0;
  if ( start == -1 )
    empty = 1;
return (empty);
}

T get(){
  int e = start;
  if ( start == end ){ // Vienas elementas
    start = -1; end = -1;
  }else
    start = previuosIndex(start);

  return (data[e]);
}
```

Eilę galime realizuoti ir vienakrypčiame tiesiniame sąraše. Tada patogu saugoti ne tik sąrašo pradžios listStart, bet ir pabaigos listEnd rodykles. Įterpdami naują elementą išvengsime visų elementų tikrinimo nuo sąrašo pradžios, kai ieškome paskutinio jo elemento.