ЗАКАРПАТСЬКИЙ УГОРСЬКИЙ ІНСТИТУТ ІМЕНІ ФЕРЕНЦА РАКОЦІ ІІ

II. RÁKÓCZI FERENC KÁRPÁTALJAI MAGYAR FŐISKOLA

Кафедра математики та інформатики
Matematika és Informatika Tanszék

АЛГОРОТМИ ТА СТРУКТУРИ ДАНИХ/ ALGORITMUSOK ÉS ADATSZERKEZETEK

МЕТОДИЧНІ ВКАЗІВКИ ДО ПРАКТИЧНИХ ЗАНЯТЬ/ MÓDSZERTANI ÚTMUTATÓ GYAKORLATI FOGLALKOZÁSOKHOZ

Алгоритми та структури даних/ Algoritmusok és adatszerkezetek
(назва навчальної дисципліни/a tantárgy neve)
Перший (бакалаврський) / Alapképzés (BSc)
(рівень вищої освіти / felsőoktatás szintje)
01 Освіта/Педагогіка / 01 Oktatás/Pedagógia
(галузь знань / képzési ág)
Середня освіта (Інформатика) / Középiskolai oktatás (Informatika)
(освітня програма / képzési program)

Берегове / Beregszász 2024 р.

Методичні вказівки «Алгоритми та структури даних» до практичних занять з дисципліни «Алгоритми та структури даних» розроблені на основі Освітньої програми підготовки бакалаврів з галузі знань «01 Освіта/Педагогіка» за напрямом «014 Середня освіта (Інформатика)» як для студентів денної, так і заочної форми навчання. Метою методичного посібника є практичне використання структур даних для типових алгоритмах обробки даних, які вивчаються в курсі «Алгоритми та структури даних». У роботі описані основні структури даних, їх властивості, а також типові задачі, для яких вони використовуються. Наведені програми розв'язування задач на мові програмування Python, а деякі задачі – на мові Pascal. Методичні вказівки можуть бути використані на практичних заняттях та при самостійній роботі по даному курсу.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 1 від 13 серпня 2024 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол №22 від «26» серпня 2024 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол №7 від « 27» серпня 2024 року)

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ

Розробник методичних вказівок:

Йожеф ГОЛОВАЧ – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор технічних наук

Рецензенти:

Олександр МІЦА – завідувач кафедри інформаційних управляючих систем та технологій УжНУ, доктор техніічних наук, професор (м. Ужгород)

Мирослав СТОЙКА – доцент кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, кандидат фізико-математичних наук, доцент

Відповідальні за випуск:

Каталін КУЧІНКА – завідувач кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доцент, кандидат фізико-математичних наук

Олександр ДОБОШ – начальник Видавничого відділу ЗУІ ім. Ф.Ракоці ІІ

За зміст методичних вказівок відповідальність несе розробник.

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua)

Az „Adatszerkezetek” módszertani utasításokat az „Algoritmusok és adatszerkezetek” tárgy gyakorlati foglalkozásaihoz a BSc „Középiskolai oktatás (Informatika)” képzési programja alapján lett kidolgozva nappali és levelező tagozatos hallgatók részére. A módszertani kézikönyv célja az adatszerkezetek gyakorlati alkalmazása tipikus adatfeldolgozási algoritmusokban, amelyeket a hallagatók az „Algoritmusok és adatstruktúrák” tárgyban tanulják. Az útmutató ismerteti a fontosabb adatstruktúrákat, azok tulajdonságait, valamint azokat a tipikus feladatokat, amelyekhez használják őket. Tartaámazza a programokat Python és Pascal programozási nyelveken, amelyek alkalmazhatók a feladatok megoldására. A módszertani utasítások használhatók a gyakorlati órákon és a tantárgy önálló munkája során.

Az oktatási folyamatban történő felhasználását jóváhagyta a
II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszéke
(2024.08.13. ,1 számú jegyzőkönyv).

Megjelentetésre javasolta a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Oktatási és Módszertani Tanácsa
(2024. augusztus 26., 22. számú jegyzőkönyv).

Elektronikus formában (PDF fájlformátumban) történő kiadásra javasolta
a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Tudományos Tanácsa
(2024. augusztus 27., 7 . számú jegyzőkönyv).

Kiadásra előkészítette a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Matematika és Informatika Tanszéke, valamint Kiadói Részlege.

A módszertani útmutató kidolgozója:

HOLOVÁCS József – professzor, műszaki tudományok doktora, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének professzora

Szakmai lektorok:

MITSA Oleksandr – Az UNE Információs vezérlési Rendszerek és Technológiák Tanszékének vezetője, műszaki tudományok doktora, professzor

SZTOJKA Miroszláv – a fizikai és matematikai tudományok kandidátusa, docens, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének docense

A kiadásért felelnek:

KUCSINKA Katalin – PhD, a fizikai és matematikai tudományok kandidátusa, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének tanszékvezetője és docense

DOBOS Sándor – a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Kiadói Részlegének vezetője

Kiadó: II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola (cím: 90 202, Beregszász, Kossuth tér 6. E-mail: foiskola@kmf.uz.ua)

TARTALOMJEGYZÉK

Előszó

Niklaus Wirth, svájci informatikus 1976-ban írt egy könyvet „Algoritmusok + adatstruktúrák = programok” címmel. 50 év elteltével ez a képlet továbbra is érvényes.

Az adatszerkezetek a számítástechnikában és a programozásban rendszerezett módon tárolják és szervezik az adatokat. Különböző típusú adatszerkezetek léteznek, melyek különböző módon alkalmasak az adatok tárolására, kezelésére és hozzáférésére.

Az adatszerkezet – egy konténer, amelyben az adatok speciális sorrendben vannak elrendezve. Ennek köszönhetően az adatstruktúra egyes műveletekben hatékony, más esetekben hatástalan lesz. Célunk, hogy az adatstruktúrákat úgy válogatjuk össze, hogy a konkrét feladathoz a legmegfelelőbbet válasszuk ki belőlük.

Ez a jegyzet elsősorban informatika szakos hallgatók számára készült, de hasznos lehet mindazok számára, akik bármely más szakon tanulják a programozást, első sorban matematikusok számára. A módszertani utasítások fejezetekre vannak osztva, és minden fejezet egy-egy adatszerkezettel foglalkozik.

Vissza

1.Adatszerkezetek típusai

- Lineáris adatszerkezetek: Lista (List), Verem (Stack), Sor (Queue).
- Nem lineáris adatszerkezetek: Fa (Tree), Gráf (Graph.)
- Egyéb speciális adatszerkezetek: Hash tábla (Hash table), Halmaz (Set).

Minden adatszerkezetnek vannak sajátosságai és előnyei, amelyek alapján kiválaszthatok az adott feladathoz vagy problémához leginkább megfelelő típusok. Az adatszerkezetek kiválasztása hatékony algoritmusok tervezéséhez és adatmanipulációhoz is kapcsolódik, hogy az adatok kezelése optimalizált és gyors legyen.

Az adatstruktúrák kiválasztása általában attól függ, hogy milyen típusú adatokat kell tárolni és milyen műveleteket kell rajtuk végrehajtani. Az alábbiakban néhány általánosan használt adatstruktúrát sorolunk fel, és bemutatjuk, milyen esetekben és milyen célokra alkalmazhatók.

1. Tömbök (масиви, array)

Felhasználás: Tömb – dinamikus adatstruktúra, amely gyors hozzáférést és manipulációt tesz lehetővé az elemekkel. Alkalmazható, ha az adatok rendezetlenek, vagy ha azok sorrendje fontos. A Pythonba a tömböket listák (Lists) által reprezentálják.

2. Láncolt listák (Зв'язні списки, **Linked List**)

[egy irányban vagy kétirányban láncolt lista]

Felhasználás: Láncolt listák hasznosak, ha gyakran szükség van beszúrási vagy törlési műveletekre a lista közepén vagy elején. Előnye, hogy nem szükséges előre meghatározni a lista méretét, és könnyen kezelhető az elemek közötti hivatkozások segítségével.

3. Vermek (Стеки, **Stack)**

Felhasználás: A vermet alkalmazzák olyan helyzetekben, amikor a legutóbbi elemeket kell először feldolgozni (LIFO - Last In, First Out elv szerint). Például mélységi keresésnél használják a rekurzió nyomon követésére.

4. Sorok (Черги, **Queue)**

Felhasználás: (Várakozási) sor alkalmazása esetén az első bejött elem kerül először feldolgozásra (FIFO - First In, First Out elv szerint). Szélességi keresésnél használható, vagy olyan esetekben, amikor fontos a műveletek sorrendje.

5. Fák (Дерева, **Trees)**

[bináris fa, bináris kereső fa, általános fa]

Felhasználás: Fa struktúrák alkalmasak hierarchikus adatok tárolására. Például bináris kereső fák használhatók rendezett adatok tárolására és gyors keresésre.

6. Gráfok (Граф, **Graphs)**

Felhasználás: Gráfok különböző kapcsolatok reprezentálására alkalmasak. Irányított és irányítatlan gráfok, súlyozott és súly nélküli élekkel is lehet őket reprezentálni.

7. Hash táblák (Хеш-таблиці, **Hash Tables)**

Felhasználás: Hash táblák gyors kulcs-érték párok tárolására és gyors hozzáférésre használhatók. Hasznosak keresési, beszúrási és törlési műveletek gyors végrehajtására.

8. Halmazok (Множини, **Sets)**

Felhasználás: Halmazok egyedi elemek gyűjteménye, amelyeket gyorsan lehet összehasonlítani és uniót, metszetet vagy különbséget végezni rajtuk.

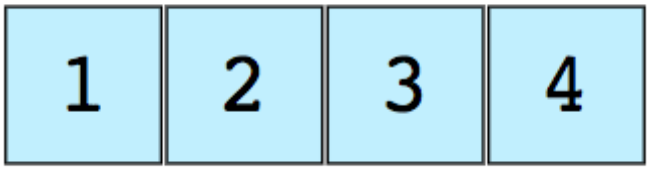
Ezek az adatstruktúrák általános esetben széles körben használatosak a programozásban, és alkalmazásuk attól függ, hogy milyen típusú adatokat kell tárolni és milyen műveleteket kell velük végezni. Az adatstruktúra kiválasztása függ a konkrét problémától és az alkalmazás követelményeitől. A konkrét feladatoktól függően az adatokat megfelelő formátumban kell tárolni. Számos adatszerkezetünk van, amelyek különböző formátumokat biztosítanak számunkra.

Vissza

2. Tömbök

A tömb a legegyszerűbb és legelterjedtebb adatstruktúra. Egyéb adatszerkezetek, például vermek és sorok, szerkeszthetők tömbök által is.

Ez egy egyszerű, 4 elemből álló tömb (1, 2, 3, 4):



Minden adatelemhez pozitív számérték van rendelve, amelyet indexnek nevezünk és a tömbelem pozícióját határozza meg. A legtöbb programnyelvben a tömbelemek 0-tól vannak számozva.

A tömb lehet:

- Egydimenziós (a fent látható módon)
- több dimenziós (az elemei tömbök)

A legegyszerűbb műveletek a tömbökkel

- *Insert* — beszúr egy elemet egy adott indexű pozícióba
- *Get* — kiadja az adott index pozícióján levő elemet
- *Delete* – egy elem törlése a megadott indexből
- *Size* – meghatározza a tömb elemeinek számát

Feladatok:

- *Keresse meg a tömb második minimális elemét*
- *Keressen egyedi egész számokat egy tömbben*
- *Összefűzni két rendezett tömböt*
- *A pozitív és negatív értékek átrendezése egy tömbben*

Vissza

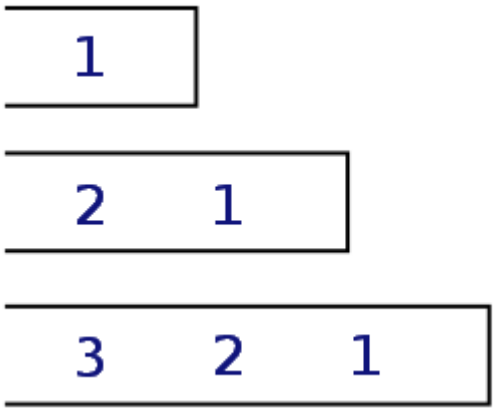
3. Vermek

Mindenki ismeri a híres "Visszavonás" (Mégse) opciót, amely szinte minden alkalmazásban megtalálható. Gondoljuk át, hogyan is működik? A jelentése a következő: a program eltárolja a munkája korábbi állapotait (a mentett állapotok száma korlátozott), és ezek a következő sorrendben helyezkednek el a memóriában: az utoljára mentett elem megy először. Egy ilyen feladat önmagában nem oldható meg tömbökkel. Itt szükség van a veremre.

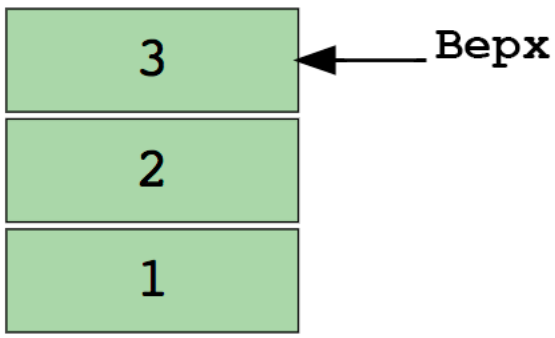
A verem egy magas köteg könyvhöz hasonlítható. Ha akarunk a köteg közepéből kivenni egy könyvet, először el kell távolítania a felette lévő összes könyvet.

A verem egy dinamikus adatstruktúra, amelyben csak az egyik végén (a legfelső (utolsó) elemen) lehet elemeket hozzáadni és eltávolítani. Ez a LIFO elv:

LIFO = Last In -> First Out (vagy FILO = First In -> Last Out)



Így néz ki egy verem, amely három adatelemet (1, 2, 3) tartalmaz, ahol a 3 van a tetején - tehát először ez lesz kiválasztva:



A veremműveletek:

A következő műveleteket hajtják végre a veremen:

- **push** (<stack_start>,<új_elem>):<verem_start>
  új elem hozzáadása a veremhez;

- **isEmpty** (<stack_start>): logikai érték
  annak meghatározása, hogy a verem üres-e;

- **pop** (<stack_start>):<stack_element_type>
  hozzáférés az utolsó beépített elemhez, a verem tetejéhez;

- **top** (<stack_start>):<stack_start>
  az utolsó benne lévő elem eltávolítása a veremből.

Veremműveletek Pythonban

Alapvető veremműveletek Python függvényekként: push, pop, top és isEmpty műveletek.

```
class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if not self.isEmpty():
            return self.items.pop()
        else:
            raise IndexError("Stack is empty")
    def top(self):
        if not self.isEmpty():
            return self.items[-1]
        else:
            raise IndexError("Stack is empty")
    def isEmpty(self):
        return len(self.items) == 0
```

Példa. class Stack alkalamzása.

```
if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack.top())   # 3
    print(stack.pop())   # 3
    print(stack.pop())   # 2
    print(stack.isEmpty())  # False
    print(stack.pop())   # 1
    print(stack.isEmpty())  # True
    stack.pop()  # Ez hibát fog eredményezni, mert a verem
üres
```

Ebben az implementációban a Stack osztály tartalmazza a verem műveleteit:

- push(item): Ezzel a metódussal egy új elemet adunk hozzá a verem tetejére.
- pop(): Ez a metódus eltávolítja és visszaadja a verem tetején lévő elemet. Ha a verem üres, hibát dob.
- top(): Ez a metódus visszaadja a verem tetején lévő elemet anélkül, hogy eltávolítaná azt. Szintén hibát dob, ha a verem üres.
- isEmpty(): Ez a metódus visszaadja, hogy a verem üres-e (True/False).

Ezek a függvények segítséget nyújtanak a verem műveletek végrehajtásához és a verem állapotának ellenőrzéséhez. Fontos, hogy a pop() és a top() műveletek előtt mindig ellenőrizzük, hogy a verem nem üres-e, hogy elkerüljük a hibás működést.

Veremműveletek Pascalban

```
Var
 a : array[1..100] of integer;
 size, x, i : integer;

procedure push(c : integer);
 begin
  size := size + 1;  a[size] := c;
 end;

 procedure pop;
 begin
  size := size - 1;
 end;

 function top : integer;
 begin
  top := a[size];
 end;

 begin
  size := 0; {Kezdetben a verem üres}
  for i := 1 to 3 do begin {a  billentyűzetről beírunk 3
elemet}
    readln(x);
    Push(x);
  end;
  writeln(Top()); {A képernyőn 9 fog megjelenni}
  while size > 0 do  {Elemek eltávolítása, amíg a verem üres
lesz}
    Pop();
  writeln(size); {A verem üres, ezért 0 kerül kinyomtatásra}
 end.
```

Verem alkalmazása aritmetikai kifejezés vizsgálatára

Írjunk be egy karakterláncot, amely tartalmaz egy aritmetikai kifejezést. Meg kell határozni, hogy a kifejezésben a zárójelek ‘(’ és ’)’ helyesen vannak-e elhelyezve (más karaktereket figyelmen kívül hagyunk).

```
( ( ) ) ()
1,2,1,0,1,0   helyes

(( ))) (
1,2,1,0,-1,0   nem helyes

(( ))(
1,2,1,0,1     nem helyes
```

Algoritmus vázlata

Az aritmetikai kifejezések helyes kiértékeléséhez zárójeleket kell megfelelően elhelyezni. Ha megfelelően használjuk a zárójeleket, akkor a verem alapú értékelés (stack-based evaluation) segíthet abban, hogy helyesen kiértékeljük a kifejezést.

1. kezdetben a verem üres;
2. ciklusban végig nézzük a sor minden karakterét;
3. ha az aktuális karakter nyitó zárójel, akkor illessze be a verem tetejére;
4. ha az aktuális karakter egy záró zárójel, akkor ellenőrizze a verem felső elemét:
ott kell Lennie a nyitó zárójelnek (ellenkező esetben hiba);
5. a végén a veremnek üresnek kell lennie, különben hiba lép fel.

Algoritmus Python nyelven

```
def helyes_kifejezes(kifejezes):
    verem = []
    for karakter in kifejezes:
        if karakter == '(':
            verem.append(karakter)
        elif karakter == ')':
            if len(verem) == 0 or verem.pop() != '(':
                return False
    return len(verem) == 0

# Példa
if __name__ == "__main__":
    kifejezes1 = "((a + b) * (c - d))"
    kifejezes2 = ")(a + b) * (c - d)("

    print(helyes_kifejezes(kifejezes1))  # True
    print(helyes_kifejezes(kifejezes2))  # False
```

Ebben a példában a *helyes_kifejezes* függvény megvizsgálja, hogy egy adott kifejezésben helyesen vannak-e elhelyezve a zárójelek. Ha minden nyitó zárójelhez van egy megfelelő lezáró zárójel, és megfelelő a sorrendjük, akkor a függvény `True`-t ad vissza, egyébként `False`-t.

Megjegyzés. Ez a példa egyszerű ellenőrzést végez a zárójelekre. Ha szükséges, összetettebb kifejezések esetén további általánositásra lehet szükség (pl. matematikai helyesség, operátorok helyessége stb.), de az alapvető struktúra ellenőrzésére ez a megközelítés hasznos lehet.

Algoritmus Pascal nyelven

```
{ veremstruktúra létrehozása }
const MAXSIZE = 50;
type Stack = record  { a verem 50 karakterre készült }
      tags: array[1..MAXSIZE] of char;
      size: integer; {elemek száma }
     end;
{ Egy elem bevitele a verembe }
procedure Push( var S: Stack; x: char);
begin
  if S.size = MAXSIZE then Exit;  //kilépés, ha verem
//túlcsordulás történik
 S.size := S.size + 1;
  S.size := S.size + 1;
  S.tags[S.size] := x; // elem hozzáadása
end;
{ eltávolítja és visszaadja a verem tetején lévő elemet }
function Pop ( var S:Stack ): char;
begin
  if S.size = 0 then begin
    Result := char(255); // 255 üres karakter, hiba, a verem
üres
    Exit;
  end;
  Result := S.tags[S.size];
  S.size := S.size - 1;
end;
{ ellenőrzés: üres-e a verem }
function isEmptyStack ( S: Stack ): Boolean;
begin
  Result := (S.size = 0);
end;
```

```
{ Fő program }
var  expr: string;
    br1, br2: char;   {zárójelek }
    i, k: integer;
    upper: char;    { a legfelső karakter kikerült a
veremből }
    error: Boolean; { hibajel}
    S: Stack;
begin
  br1 := '('; br2 := ')';
  S.size := 0;
  error := False;
writeln('Írja be a szöveget'); readln(expr);...
for i:=1 to length(expr) do { karakterlánc-feldolgozású
ciklus }
  begin
      if expr[i] = br1 then begin { nyitó zárójel '(' }
        Push(S, expr[i]); { verembe tolja }
        break;
   end;
    if expr[i] = br2  then begin { záró zárójel ')' }
      upper := Pop(S); { szimbólum kiemelése a veremből }
      error := upper <> br1; { a verem üres vagy rossz a
zárójel }
        break;
      end;
    if error then break; // Hiba történt
end;
```

Feladat. A veremben tárolt adatok rendezése.

A verem alapvetően LIFO (Last In, First Out) elven működik.

Ezért a verem alapvetően nem támogatja a rendezést anélkül, hogy közben megváltoztatnánk annak alapvető működését. Azonban, ha szükség van a veremben lévő elemek rendezésére, akkor külön kell tárolni az elemeket, rendezni azokat, majd vissza kell tölteni a rendezett sorrendben egy új verembe, vagy a verem eredeti elemeit felül kell írni a rendezett elemekkel.

Python-program, amely rendezi egy veremben lévő elemeket:

```
def rendez_verem(stack):
    # Kimentjük a verem elemeit egy ideiglenes listába
    temp_list = []
    while not stack.isEmpty():
        temp_list.append(stack.pop())
    # Rendezzük az ideiglenes listát (feltehetjük, hogy
növekvő sorrendben szeretnénk)
    temp_list.sort()
    # Visszatöltjük a rendezett elemeket a verembe
    for elem in temp_list:
        stack.push(elem)
```

A program alkalmazása:

```
if __name__ == "__main__":
    stack = Stack()
    stack.push(5)
    stack.push(2)
    stack.push(7)
    stack.push(3)
    # Rendezzük a veremben lévő elemeket
    rendez_verem(stack)
    # Kiírjuk a rendezett elemeket
    while not stack.isEmpty():
        print(stack.pop())  # Ez 2, 3, 5, 7 lesz
```

Ebben a példában a *rendez_verem* függvény először kiveszi a verem összes elemét egy ideiglenes listába (*temp_list*). Ezután a *sort()* metódussal rendezzük az ideiglenes listát. Végül visszatöltjük a rendezett elemeket újra a verembe, így azok növekvő sorrendben lesznek.

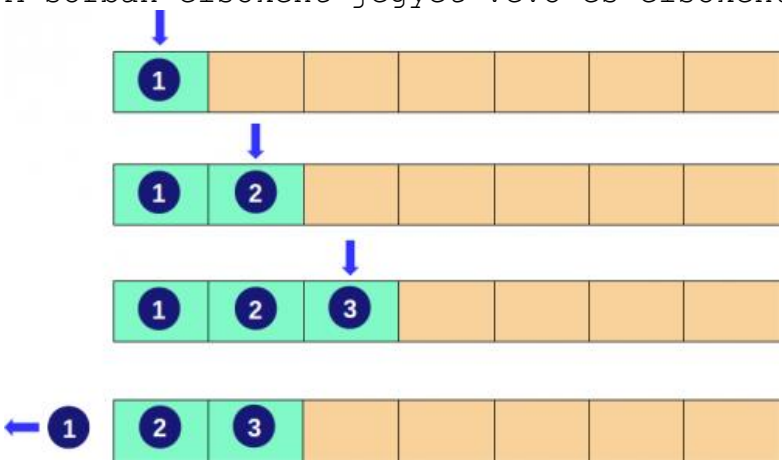
Megjegyzés. Ez a megoldás csak akkor működik, ha a verem elemei rendezhetők, és a rendezés során az eredeti verem struktúráját nem kell megváltoztatni.

Vissza

4. Sorok

A sor, akárcsak a verem, egy lineáris adatstruktúra, amelyben az elemek szekvenciális sorrendben kerülnek tárolásra. Az egyetlen lényeges különbség a verem és a sor között, hogy a LIFO helyett a FIFO (First In - First Out) elve működik a sorban. Egy rövidítés a sor kifejezésre: FIFO = First In – First Out.

A sorbaállás reális példája az ügyfelek sorban állása a jegypénztárnál. Az új vevő a sor vége lesz, nem pedig a kezdet. A sorban elsőként jegyet vevő és elsőként távozó lesz.



A sor egy dinamikus adatstruktúra, amelyben mindig csak két elem érhető el: az első és az utolsó. Elemek hozzáadása csak az egyik végéről (a sor végéről), az elemek törlése pedig csak a másik végéről (a sor elejéről) lehetséges.

A következő műveletek állnak rendelkezésre a sornál:
- **PushTail** – egy elem hozzáadása a sor végéhez (vagy Enqueue);
- **Pop** – egy elem eltávolítása a sor elejéről (vagy Dequeue).
- **isEmpty()** – Igaz értéket ad vissza, ha a sor üres
- **Top()** – A sor első elemét adja vissza

Sorműveletek megvalósítása tömbök alkalmazásával

Ez egy meglehetősen egyszerű módszer, amelynek két hátránya van: a tömb előzetes kiosztása, az elemek eltolása a sorból való eltávolításukkor.

Pythonban a listát használhatjuk egy sor implementálására, de fontos figyelembe venni, hogy a lista alapvetően dinamikus tömbként működik, és az elemeihez történő hozzáférés hatékonyan történik mind az elején, mind a végén.

Sor műveletek megvalósítása Pythonban (listával)

```
# Sor műveletek (listával):
class Sor:
    def __init__(self):
        self.elements = []

    def ures(self):
        return len(self.elements) == 0

    def sorba_allit(self, elem):
        self.elements.append(elem)

    def sorbol_kivesz(self):
        if self.ures():
            raise IndexError("A sor üres, nem lehet elemet
kivenni.")
        return self.elements.pop(0)

    def elso(self):
        if self.ures():
            raise IndexError("A sor üres, nincs első elem.")
        return self.elements[0]

    def meret(self):
       return len(self.elements)
```

Program alkalmazása

```
if __name__ == "__main__":
    sor = Sor()

    sor.sorba_allit(1)
    sor.sorba_allit(2)
    sor.sorba_allit(3)
    print("Elso elem:", sor.elso())  # Elso elem: 1
    print("Sorból kivett elem:", sor.sorbol_kivesz())  #
Sorból kivett elem: 1
    sor.sorba_allit(4)
    print("Sor mérete:", sor.meret())  # Sor mérete: 3
    while not sor.ures():
        print("Kivett elem:", sor.sorbol_kivesz())  # Kivett
elemek: 2, 3, 4
        return len(self.elements)
```

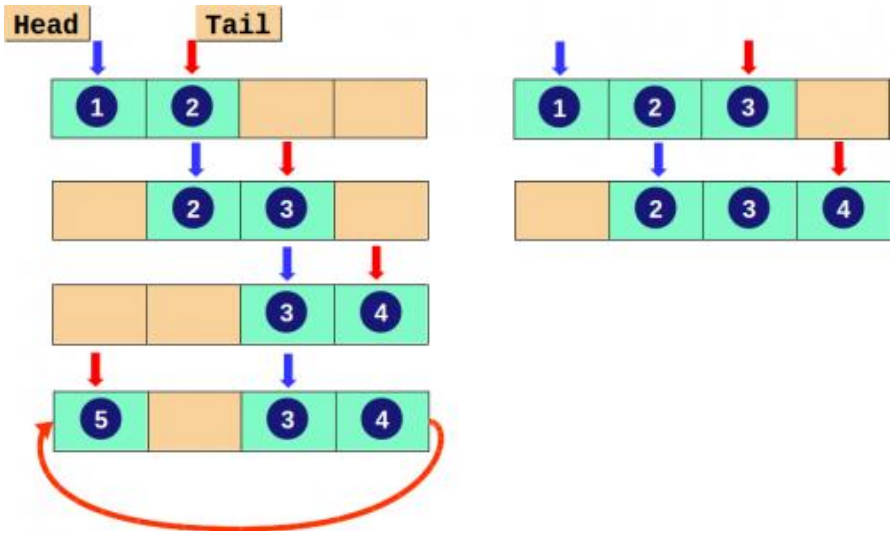
Ebben a példában a Sor osztály egy sor műveleteit valósítja meg:

- ures(): Ellenőrzi, hogy a sor üres-e.
- sorba_allit(elem): Hozzáad egy új elemet a sor végére.

- sorbol_kivesz(): Kiveszi és visszaadja a sor elején lévő elemet. Ha a sor üres, hibát dob.
- elso(): Visszaadja a sor elején lévő elemet anélkül, hogy eltávolítaná azt. Ha a sor üres, hibát dob.
- meret(): Visszaadja a sorban lévő elemek számát.

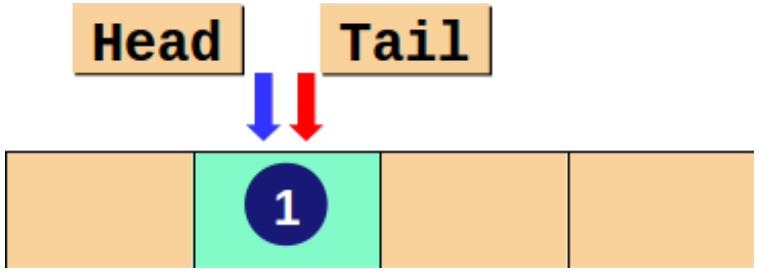
Ez a példa demonstrálja, hogyan lehet felhasználni egy listát arra, hogy sor műveleteket valósítsunk meg Pythonban. A listák rugalmassága miatt ez a megoldás egyszerű, de hatékony módot nyújt a FIFO adatszerkezet megvalósítására.

Sorműveletek gyűrűtömb használatával

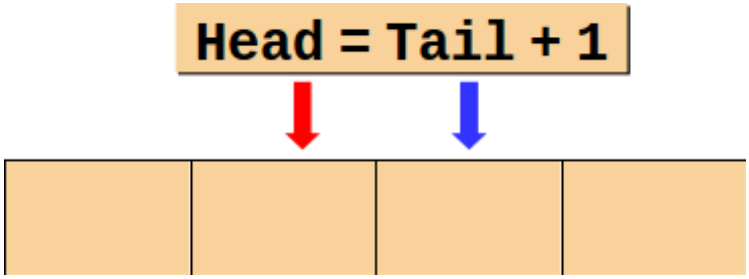


Esetek

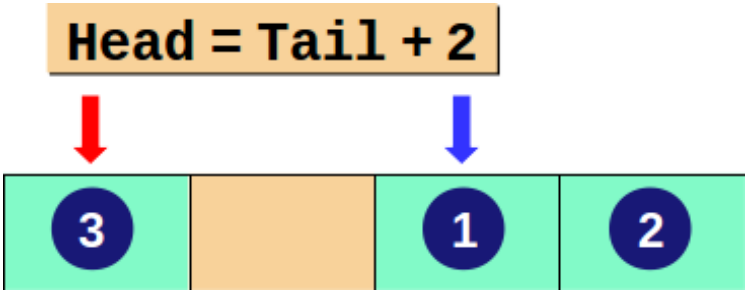
A sor 1 elemet tartalmazz:



A sor üres:



A sor megtelt:



A gyűrűtömb segítségével lehet megvalósítani egy sort. A gyűrűtömb lehetővé teszi az elemek hatékony hozzáadását a sor végéhez, és az elemek eltávolítását a sor elejéről, miközben gyors hozzáadási és eltávolítási műveleteket biztosít.

Gyűrűtömb alkalmazása. Pyton

```
# Gyűrűtömb 1
class CircularQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0  # A sor elejének indexe
        self.rear = 0   # A sor végének indexe
        self.size = 0   # A sor aktuális mérete
    def is_empty(self):
        return self.size == 0
    def is_full(self):
        return self.size == self.capacity
    def enqueue(self, item):
        if self.is_full(): raise Exception("Queue is full")
                self.queue[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity
        self.size += 1
   def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
                item = self.queue[self.front]
        self.queue[self.front] = None
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return item
```

```
# Gyűrűtömb 2
# class CircularQueue:

    def peek(self):
        if self.is_empty(): raise Exception("Queue is
empty")
        return self.queue[self.front]
    def display(self):
        print("Current Queue:")
        if self.is_empty():   print("Empty")
        else:
            for i in range(self.size):
                print(self.queue[(self.front + i) %
self.capacity], end=" ")
            print()
```

Program alkalmazása:

```
cq = CircularQueue(5)
cq.enqueue(1)
cq.enqueue(2)
cq.enqueue(3)
cq.enqueue(4)
cq.display()  # Output: Current Queue: 1 2 3 4
print("Dequeuing:", cq.dequeue())  # Output: Dequeuing: 1
cq.enqueue(5)
cq.enqueue(6)  # A sor megint megtelt
cq.display()  # Output: Current Queue: 2 3 4 5 6
```

A program magyarázata

Inicializálás.

Az __init__ konstruktor inicializál egy gyűrűtömböt (self.queue), adott kapacitással (self.capacity), indexeli a self.front és self.rear-t. nyomon követheti a sor kezdetét és végét, valamint a „self.size” értéket a sor aktuális méretének nyomon követéséhez.

Metodusok.

Az is_empty() és is_full() ellenőrzi, hogy a sor üres-e vagy megtelt.

Az enqueue(item) egy elemet ad a sorhoz, növeli a self.rear értéket, és biztosítja a hurkot, amikor eléri a tömb végét.

A dequeue() lekéri az elemet a sor elejéről, növeli a self.front értéket, és hurkot is biztosít.

A peek() a sor elején lévő elemet adja vissza anélkül, hogy eltávolítaná azt.

A display() megjeleníti a sor aktuális tartalmát.

Ez a példa egy alapvető várólista-megvalósítást mutat be Python gyűrűtömb használatával. Valós használatban a funkcionalitás finomítható kivételkezelés és szükség szerint további metódusok hozzáadásával.

Sorműveletek gyűrűtömb alkalamazásával. Pascal

```
{ gyűrűtömb tipus létrehozása }
type Queue = record
       data: array[1..MAXSIZE] of integer;
       head, tail: integer;
     end;
{ Elem hozzáadása a sorhoz }

procedure PushTail( var Q: Queue; x: integer);
begin
  if Q.head = (Q.tail+1) mod MAXSIZE + 1
    then Exit; { a sor megtelt }
  Q.tail := Q.tail mod MAXSIZE + 1;
  Q.data[Q.tail] := x;
end;

{ elem eltávolítása a sor elejéről (vagy Dequeue)}
function Pop ( var S: Queue ): integer;
begin
  if Q.head = Q.tail mod MAXSIZE + 1 then begin
    Result := MaxInt;
    Exit;
  end;
  Result := Q.data[Q.head];
  Q.head := Q.head mod MAXSIZE + 1;
end;
```

Vissza

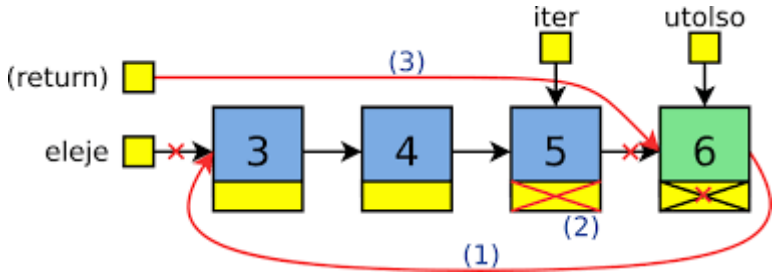
5. Láncolt listák

A Láncolt lista (linklista) fontos lineáris adatstruktúra, amely első pillantásra egy tömbre hasonlít. A láncolt lista azonban különbözik a tömbtől a memóriafoglalásban, a belső felépítésben, valamint abban, hogy az alapvető beillesztési és törlési műveleteket hogyan hajtják végre benne.

A láncolt lista csomópontok láncára hasonlít, amelyek mindegyike információt tartalmaz: adatokat és egy mutatót a lánc következő csomópontjára. A lista első elemének megfelelő fő mutató található, és ha a lista üres, akkor egyszerűen nullára (null) mutat.

A láncolt listák fájlrendszerek, hash táblák és szomszédsági listák megvalósítására szolgálnak.

Így néz ki a láncolt lista belső szerkezetét:



A következő típusú láncolt listákat alkalmaznak:

- **egyirányú láncolt lista**
- **kétirányú láncolt lista.**

Műveletek láncolt listákkal:

- **InsertAtEnd** – A megadott elemet beszúrja a lista végére
- **InsertAtHead** – A megadott elemet beszúrja a lista elejére
- **Delete** – Az adott elem törlése a listából
- **DeleteAtHead** – Töröli az első elemet a listából
- **Search** – A megadott elemet adja vissza a listából
- **isEmpty** – Igaz értéket ad vissza, ha a lista üres

Láncolt lista megvalósítása Pythonban

Láncolt lista megvalósítása során általában egy `Node` (csomópont) osztályt definiálunk, amely egy adatelemet (értéket) és egy referencia-tulajdonságot (pointer-t) tartalmaz a következő csomópontra. Ezután létrehozunk egy `LinkedList` (láncolt lista) osztályt, amely kezeli és szolgáltatja a láncolt lista műveleteit: elemek beszúrását, törlését, keresését, stb.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def is_empty(self):
        return self.head is None
    def append(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
    def prepend(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node
def delete_value(self, value):
    if self.head is None:  return
    if self.head.value == value:
       self.head = self.head.next:
       return
        current_node = self.head
        while current_node.next:
            if current_node.next.value == value:
                current_node.next = current_node.next.next
                return
            current_node = current_node.next
    def search(self, value):

        current_node = self.head
        while current_node:
            if current_node.value == value:
                return True
            current_node = current_node.next
        return False
    def print_list(self):
        current_node = self.head
        while current_node:
            print(current_node.value, end=" -> ")
            current_node = current_node.next
        print("None")
```

Algoritmus alkalmazására:

```
if __name__ == "__main__":
    linked_list = LinkedList()
    linked_list.append(1)
    linked_list.append(2)
    linked_list.append(3)
    linked_list.print_list()  # 1 -> 2 -> 3 -> None
    linked_list.prepend(0)
    linked_list.print_list()  # 0 -> 1 -> 2 -> 3 -> None
    linked_list.delete_value(2)
    linked_list.print_list()  # 0 -> 1 -> 3 -> None
    print("Search 1:", linked_list.search(1))  # Search 1:
_
```

Node osztály
Node: Ez az osztály egy csomópontot reprezentál a láncolt listában. A value tulajdonság tárolja az adatelemet, míg a next tulajdonság egy referencia a következő csomópontra.

LinkedList osztály
__init__: Létrehozza a láncolt lista fejét (head). Kezdetben ez None, ami azt jelzi, hogy a lista üres.
is_empty: Ellenőrzi, hogy a lista üres-e.
append: Hozzáad egy új csomópontot a lista végére.
prepend: Hozzáad egy új csomópontot a lista elejére.
delete_value: Töröli az adott értéket tartalmazó csomópontot a listából.
search: Keresés az adott értéket tartalmazó csomópontban.
print_list: Kiírja a lista elemeit sorban.

Ez a példa mutatja, hogyan lehet létrehozni és használni egy egyszerű láncolt listát Pythonban. A láncolt listák különösen hasznosak akkor, ha gyakran szükség van elemek beszúrására vagy törlésére a lista közepén, mivel ezek a műveletek hatékonyan végezhetők el a pointerek (mutatok) segítségével.

Feladatok:
- Invertálni a láncolt listát (fordított sorrend)
- Keressen egy hurkot a láncolt listában
- A lista első eleme legyen a lista N-ik eleme
- Távolítsa el az ismétlődő értékeket a láncolt listából

Vissza

6. Gráfok

A gráf olyan csomópontok halmaza, amelyek hálózat formájában kapcsolódnak egymáshoz. A csomópontokat csúcsoknak is nevezik. A gráfok csomópontok (vertexek) és az ezeket összekötő élek (élek) halmazaiként definiálhatók.

A gráfokat, mint matematikai struktúrákat, általában különböző módon lehet megvalósítani, attól függően, hogy *irányított* vagy *irányítatlan* gráfról van szó, és hogy *súlyozott* vagy *súlyozatlan* gráfról beszélünk.

Grafok osztályozása:

- irányított súlyozott gráf
- irányított súlyozatlan gráf
- irányítatlan súlyozott gráf
- irányítatlan súlyozatlan gráf

Gráfok reprezentálása

1. Szomszédsági lista (Adjacency List)

A szomszédsági lista a gráf tárolásának egyik gyakori módja, amely a gráfot úgy tárolja, hogy minden csomópontjához egy listát rendel, amely tartalmazza azokat a csomópontokat, amelyekkel közvetlenül összekötve van.

Python program.

```
     class Graph:
         def __init__(self):
             self.adjacency_list = {}

         def add_vertex(self, vertex):
             if vertex not in self.adjacency_list:
                 self.adjacency_list[vertex] = []

         def add_edge(self, vertex1, vertex2):
             if vertex1 in self.adjacency_list and vertex2
in self.adjacency_list:
     self.adjacency_list[vertex1].append(vertex2)
     self.adjacency_list[vertex2].append(vertex1)  # Ha
irányítatlan gráfról van szó
    def get_neighbors(self, vertex):
             if vertex in self.adjacency_list:
                 return self.adjacency_list[vertex]
             else:
                 return []

         def __str__(self):
             return str(self.adjacency_list)
```

Program alkalmazása:

```
      if __name__ == "__main__":
         graph = Graph()
         graph.add_vertex('A')
         graph.add_vertex('B')
         graph.add_vertex('C')
         graph.add_vertex('D')
         graph.add_edge('A', 'B')
         graph.add_edge('A', 'C')
         graph.add_edge('B', 'D')
         graph.add_edge('C', 'D')
         print("Szomszédsági lista:")
         print(graph)  # {'A': ['B', 'C'], 'B': ['A', 'D'],
'C': ['A', 'D'], 'D': ['B', 'C']}
         print("Szomszédok lekérése:")
         print("Szomszédok 'A' csomóponthoz:",
graph.get_neighbors('A'))  # ['B', 'C']
         print("Szomszédok 'D' csomóponthoz:",
graph.get_neighbors('D'))  # ['B', 'C']
```

2. Szomszédsági mátrix (Adjacency Matrix)

A szomszédsági mátrix egy másik módja a gráf tárolásának, amely egy mátrixot használ, hogy reprezentálja a gráf éleinek jelenlétét vagy hiányát.

```
     class Graph:
         def __init__(self, size):
             self.size = size
             self.adjacency_matrix = [[0] * size for _ in
range(size)]
         def add_edge(self, vertex1, vertex2):
             if vertex1 < self.size and vertex2 < self.size:
                 self.adjacency_matrix[vertex1][vertex2] = 1
                 self.adjacency_matrix[vertex2][vertex1] = 1
# Ha irányítatlan gráfról van szó
         def print_matrix(self):
             for row in self.adjacency_matrix:
                 print(row)
```

Példa használat:

```
     if __name__ == "__main__":
         graph = Graph(4)
         graph.add_edge(0, 1)
         graph.add_edge(0, 2)
         graph.add_edge(1, 3)
         graph.add_edge(2, 3)
         print("Szomszédsági mátrix:")
         graph.print_matrix()
```

Kimenet:

```
[0, 1, 1, 0]
[1, 0, 0, 1]
[1, 0, 0, 1]
[0, 1, 1, 0]
```

Gráf osztályai és tulajdonságai

A gráfok megvalósításának másik aspektusa az osztályok és tulajdonságok definiálása a Pythonban:

Graph (Gráf): Az alapvető osztály, amely tartalmazza a gráf adatszerkezetét és műveleteit.

Vertex (Csomópont): Az osztály, amely egy gráf csomópontját reprezentálja. Tartalmazhat egyedi azonosítót és egyéb tulajdonságokat.

Edge (Él): Az osztály, amely egy gráf élét reprezentálja, tartalmazhat súlyt és irányítást (ha szükséges).

Összegzés

A gráfokat Pythonban különböző módon lehet megvalósítani, attól függően, hogy milyen típusú gráfról van szó és hogy milyen műveleteket szeretnénk rajta végezni. A szomszédsági lista és szomszédsági mátrix – két gyakori módszer a gráfok tárolására, és mindegyiknek vannak előnyei és hátrányai különböző alkalmazásokban.

Egy élnek lehet súlya/költsége – egy mutató, amely azt jellemezi, hogy mennyire költséges az x csúcsból az y csúcsba való átmenet.

Általános gráfbejárási algoritmusok:

- Keresés szélességben
- Keresés mélységben

Feladat. Adva egy gráf. Határozza meg, hogy ez a gráf fa-e

A gráf fa tulajdonsága ellenőrizhető néhány algoritmus segítségével.

Egy gráf akkor és csak akkor tekinthető fának, ha az *összefüggő* (connected) és nincsenek benne *körök*, *ciklusok* (acyclic).

Algoritmusok a gráf fatisztaságának ellenőrzésére:

1. *Összefüggések ellenőrzése*

Szükséges feltétele annak, hogy egy gráf legyen tekinthető fának, ha összefüggő, azaz bármely két csúcs között létezik út.

2. Körök ellenőrzése

Egy másik alapvető tulajdonság, hogy egy gráf akkor lesz fa, ha nincsenek benne körök.

Program. Python

```
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adjacency_list = {vertex: [] for vertex in
vertices}
    def add_edge(self, u, v):
        self.adjacency_list[u].append(v)
        self.adjacency_list[v].append(u)  # Ha irányítatlan
gráf
    def is_cyclic_util(self, v, visited, parent):
        visited[v] = True
        for neighbour in self.adjacency_list[v]:
            if not visited[neighbour]:
                if self.is_cyclic_util(neighbour, visited,
v):
                    return True
            elif parent != neighbour:
                return True
        return False
    def is_tree(self):
        visited = {v: False for v in self.vertices}
        if self.is_cyclic_util(self.vertices[0], visited, -
1):
            return False
        for v in self.vertices:
            if not visited[v]:
                return False
        return True
```

Példa használat:

```
if __name__ == "__main__":
    vertices = ['A', 'B', 'C', 'D']
    graph = Graph(vertices)
    graph.add_edge('A', 'B'); graph.add_edge('A', 'C')
    graph.add_edge('B', 'D'); graph.add_edge('C', 'D')
    if graph.is_tree():    print("A gráf fa.")
    else:         print("A gráf nem fa.")
```

Algoritmus magyarázata

1. Graph osztály: Ez a gráf struktúrát kezeli. Az `add_edge` függvényel élek adhatók hozzá a gráfhoz.
2. is_cyclic_util: Ez a függvény rekurzívan ellenőrzi, hogy van-e kör a gráfban. Ha egy csomópontot újra megpróbálnak elérni, és az már korábban meglátogatott csúcsból származik, akkor a gráf tartalmaz egy kört.

3. is_tree: Ez a függvény ellenőrzi, hogy a gráf fa-e. Először ellenőrzi a gráf összefüggőségét a `is_cyclic_util` segítségével, majd azt is ellenőrzi, hogy minden csúcsot meglátogattak-e.

Az itt bemutatott példa az egyszerűség kedvéért irányítatlan és súly nélküli gráfokat vesz figyelembe. Az irányított gráfok vagy a súlyozott gráfok fatisztaságának ellenőrzéséhez további módosításokra lenne szükség a kódhoz.

Feladat. Keressük meg a legrövidebb utat két csúcs között

A legrövidebb út keresése két csúcs között egy gráfban általában a keresési algoritmusok segítségével történik, például a szélességi keresés (Breadth-First Search, BFS) vagy a Dijkstra algoritmus. Ezek az algoritmusok megfelelően alkalmazhatók irányítatlan és súlyozott gráfokra is, attól függően, hogy szükség van-e súlyozásra (pl. élek költségei) vagy sem.

Szélességi keresés (BFS) alkalmazása legrövidebb út keresésére

A szélességi keresés egy egyszerű, de hatékony algoritmus a legrövidebb út keresésére, ha a gráfban minden él ugyanannyi költséggel bír (pl. minden él súlya 1).

```
from collections import deque
class Graph:
    def __init__(self):
        self.adjacency_list = {}
    def add_edge(self, u, v):
        if u not in self.adjacency_list:
            self.adjacency_list[u] = []
        if v not in self.adjacency_list:
            self.adjacency_list[v] = []
        self.adjacency_list[u].append(v)
        self.adjacency_list[v].append(u)  # irányítatlan
gráf
    def shortest_path(self, start, end):
        queue = deque([(start, [start])])
        visited = set([start])
        while queue:
            (vertex, path) = queue.popleft()
            for neighbour in self.adjacency_list[vertex]:
                if neighbour not in visited:
                    if neighbour == end:
                        return path + [neighbour]
                    else:
                        visited.add(neighbour)
                        queue.append((neighbour, path +
[neighbour]))
        return None  # Ha nem található útvonal
```

Példa használat:

```
if __name__ == "__main__":
    graph = Graph()
    graph.add_edge('A', 'B');  graph.add_edge('A', 'C')
    graph.add_edge('B', 'D');  graph.add_edge('C', 'D')
    graph.add_edge('D', 'E')
    start = 'A';     end = 'E'
    shortest_path = graph.shortest_path(start, end)
    if shortest_path:
        print(f"A legrövidebb út {start} és {end} között: {'
-> '.join(shortest_path)}")
    else:
        print(f"Nincs út {start} és {end} között.")
```

Algoritmus magyarázata:

Ez az algoritmus a szélességi keresés alapján találja meg a legrövidebb utat, feltételezve, hogy minden él ugyanannyi költséggel rendelkezik.

1. Graph osztály: Ez a gráf struktúrát kezeli. Az `add_edge` függvénnyel élek adhatók hozzá a gráfhoz.

2. shortest_path: Ez a függvény a szélességi keresés alapján keresi meg a legrövidebb utat a `start` és `end` csúcsok között. Használ egy FIFO sor (`queue`) struktúrát a csúcsok feldolgozására és egy halmazt (`visited`), hogy nyomon kövesse, mely csúcsokat látogatták meg már.

3. Példa használat: Egy egyszerű példa bemutatása, ahol az `A` és `E` csúcsok közötti legrövidebb utat keresünk.

Feladatok:

• Hajtsa végre a szélesség és a mélység szerinti keresést a gráfon

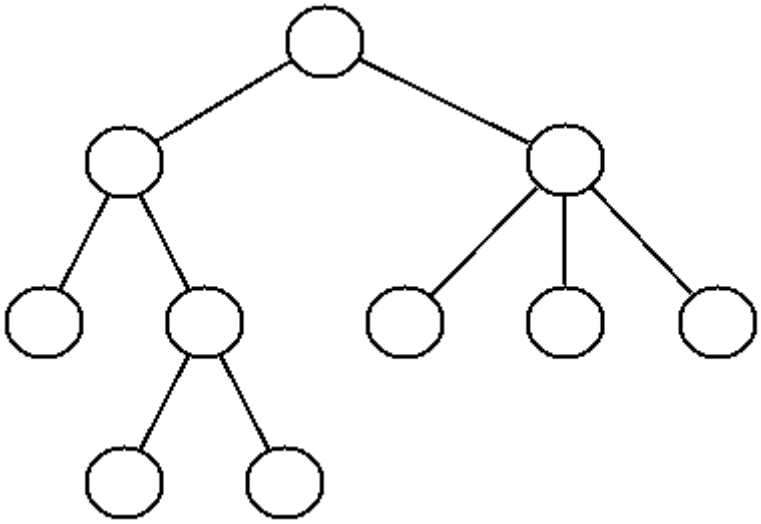
• Számolja meg a gráf éleinek számát

Vissza

7. Fák

A fa egy hierarchikus adatstruktúra, amely csúcsokból (csomópontokból) és az őket összekötő élekből áll. A fák hasonlóak a gráfokhoz, azonban a legfontosabb különbség a fa és a gráf között az, hogy a fában nincsenek ciklusok.

A fákat széles körben használják a mesterséges intelligencia területén és összetett algoritmusokban, amelyek hatékony információtárként szolgálnak a problémák megoldása során.

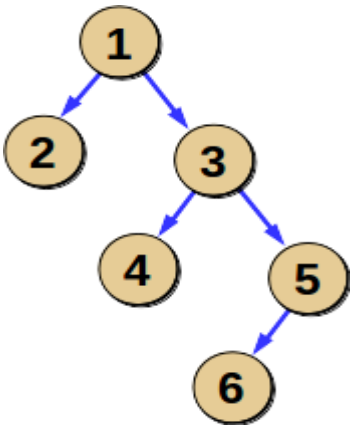


A következő típusú fákat alkalmaznak:

- N-ágú fa;
- Kiegyensúlyozott fa;
- Bináris fa;

• Bináris keresőfa;
• AVL-fa;
• Vörös-fekete fa;
• 2-3 fa.

A fent felsorolt fák közül leggyakrabban a bináris fa és a bináris keresőfa használatos.
• A gyökér a fa fő vagy kezdő csomópontja.
• A levél olyan csomópont, amelyből nem jön ki ív
• A fa magassága a gyökér és a levél közötti legnagyobb távolság (az ívek száma).



.

Néhány hierarchikus kapcsolat a példa-fában:
1 – a fa összes többi csomópontjának őse
6 – 5, 3, 1 csomópontok utódja
3 – 4, 5 csomópontok szülője
5 – 3 csomópont gyermeke
5 – 4 csomópont testvére

Bináris fa

A bináris fa (binary tree) egy fontos adatszerkezet, melyet széles körben használnak az informatikában, például keresési és rendezési algoritmusokban. A bináris fa mindegyik csomópontja legfeljebb két gyermekcsomópontot tartalmazhat: bal és jobb gyermeket. A bináris fa műveletei közé tartoznak:

- új elem hozzáadása (beszúrás)
- keresés
- törlés
- fa bejárása

Bináris fa. Alapvető műveletek. Megvalósítás Pythonban

Létrehozunk egy egyszerű bináris fa osztályt, amely képes létrehozni egy fát, új elemeket hozzáadni, keresni egy adott elemet, és bejárni a fát.

```
class treenode:
     def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
class BinaryTree:
    def __init__(self):    self.root = None
    def insert(self, key):
        if self.root is None:
             self.root = TreeNode(key)
        else:
             self._insert_recursive(self.root, key)
    def _insert_recursive(self, root, key):
        if key < root.val:
            if root.left is None:
                root.left = TreeNode(key)
            else:
               self._insert_recursive(root.left, key)
        else:
            if root.right is None:
                     root.right = TreeNode(key)
            else:
                    self._insert_recursive(root.right, key)
    def search(self, key):
        return self._search_recursive(self.root, key)
    def _search_recursive(self, root, key):
        if root is None or root.val == key:
            return root
        if key < root.val:
            return self._search_recursive(root.left, key)
        else:
            return self._search_recursive(root.right, key)

    def inorder_traversal(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result
    def _inorder_recursive(self, root, result):
        if root:
            self._inorder_recursive(root.left, result)
            result.append(root.val)
            self._inorder_recursive(root.right, result)
```

Példa.

```
bt = BinaryTree()
bt.insert(50)
bt.insert(30)
bt.insert(20)
bt.insert(40)
bt.insert(70)
bt.insert(60)
bt.insert(80)

print("Inorder bejárás eredménye:", bt.inorder_traversal())
# Output: [20, 30, 40, 50, 60, 70, 80]

search_result = bt.search(40)
if search_result:
    print("Az 40 elem megtalálható a fában.")
else:
    print("Az 40 elem nem található meg a fában.")
```

Algoritmus magyarázata

1. **TreeNode osztály**: Egy csomópontot reprezentál a bináris fában. Minden csomópontnak van egy értéke (`val`), valamint bal (`left`) és jobb (`right`) gyermek mutatója.
2. **BinaryTree osztály**: Egy bináris fa osztályt definiál, amely tartalmazza a gyökérelemet (`root`).
   - `insert(key)`: Új elem beszúrása a fába.
   - `_insert_recursive(root, key)`: Rekurzív segédmetódus az elem beszúrásához.
   - `search(key)`: Keresés az adott elem alapján.
   - `_search_recursive(root, key)`: Rekurzív segédmetódus a kereséshez.
   - `inorder_traversal()`: Inorder bejárás (bal-gyökér-jobb).
   - `_inorder_recursive(root, result)`: Rekurzív segédmetódus az inorder bejárás implementálásához.
3. **Példa**. Egy példafát hozunk létre `bt` objektumként, beszúrunk néhány elemet (`insert`), majd elvégezzük az inorder bejárását (`inorder_traversal`). Végül keresünk egy elemet (`search`) és kiírjuk, hogy az adott elem megtalálható-e a fában.

Ez a Python-program egy egyszerű bináris fa implementációt mutat be Pythonban, amely képes alapvető műveletek végrehajtására. A bináris fa tovább fejleszthető többi bejárási móddal (preorder, postorder) és további műveletekkel (például törlés).

Bináris fa alkalmazása. Pascal

Pascalban bináris fát rekordstruktúra vagy osztály használatával lehet megvalósítani. Példa egy bináris fa egyszerű megvalósítására Pascalban.

```
program BinaryTree;
type
  PNode = ^TNode;
  TNode = record
    Data: Integer;
    Left, Right: PNode;
  end;
// Függvény egy új bináris fa csomópont létrehozásához
function NewNode(Data: Integer): PNode;
var
  Node: PNode;
begin
  New(Node);
  Node^.Data := Data;
  Node^.Left := nil;
  Node^.Right := nil;
  Result := Node;
end;
// Függvény új csomópont beszúrására egy bináris fába
function Insert(Node: PNode; Data: Integer): PNode;
begin
  if Node = nil then
    Result := NewNode(Data)
  else
  begin
    if Data < Node^.Data then
      Node^.Left := Insert(Node^.Left, Data)
    else if Data > Node^.Data then
      Node^.Right := Insert(Node^.Right, Data);
    Result := Node;
  end;
end;
```

Példa. Bináris fa alkalmazása

```
var
  Root: PNode;
begin
  Root := nil;
// Elemek beillesztése a fába
  Root := Insert(Root, 50);
  Insert(Root, 30);  Insert(Root, 20);
  Insert(Root, 40);  Insert(Root, 70);
  Insert(Root, 60);  Insert(Root, 80);
// Ezután a fa bejárása (például inorder, preorder,
postorder),
 // csomópontok keresése, csomópontok törlése és egyéb
műveletek  // Memória felszabadítása (a fa törlése)
  Dispose(Root);
end.
```

Ez a példa egy bináris fa alapstruktúráját mutatja be beszúrási és felszabadítási függvényekkel. A beillesztés a csomópont értékétől függően rekurzív módon történik. A valós alkalmazásokban további funkciókat is hozzáadhat, például csomópontkeresést, fabejárásokat (inorder, preorder, postorder) és egyéb műveleteket a feladat követelményeitől függően.

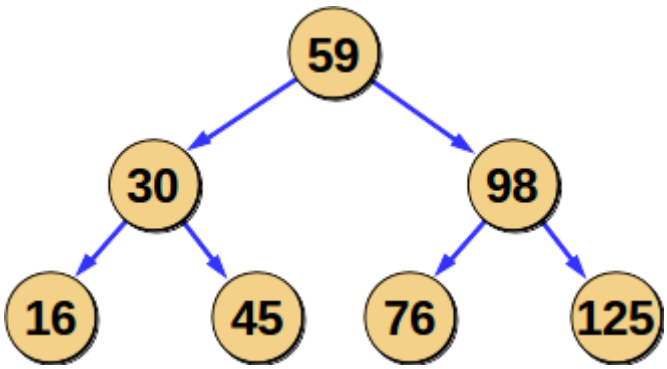
Csomópontkeresés

A fában való kereséskor a kulcs fogalmát használjuk. A kulcs annak a csomópontnak a jellemzője, amellyel a keresést végrehajtják. A kisebb kulcsú csomópontok balra, a nagyobb kulcsú csomópontok pedig jobbra mennek.

Feladat:

Keressen egy k-val egyenlő kulcsot a fában.
Algoritmus vázlata:

- ha a fa üres, az azt jelenti, hogy a kulcs nem található;
- ha a csomópont kulcsa egyenlő k-val, akkor stop;
- ha a csomópont kulcsa kisebb, mint k, akkor k keresése a bal oldali részfában;
- ha a csomópont kulcsa nagyobb, mint k, akkor a k keresése a jobb oldali részfában.



Keresés bináris fában. Python

Ha bináris fában szeretnénk keresni egy olyan kulcsot, ami egy adott értékkel (`k`) egyenlő, akkor az alábbiak szerint tehetjük meg a keresést Pythonban. A bináris fa implementációját használjuk:

```
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

class BinaryTree:
    def __init__(self):
        self.root = None
    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert_recursive(self.root, key)
    def _insert_recursive(self, root, key):
        if key < root.val:
            if root.left is None:
                root.left = TreeNode(key)
            else:
                self._insert_recursive(root.left, key)
        else:
            if root.right is None:
                root.right = TreeNode(key)
            else:
                self._insert_recursive(root.right, key)
    def search(self, key):
        return self._search_recursive(self.root, key)
    def _search_recursive(self, root, key):
        if root is None or root.val == key:
            return root
        if key < root.val:
            return self._search_recursive(root.left, key)
        else:
            return self._search_recursive(root.right, key)
```

Program alkalmazása:

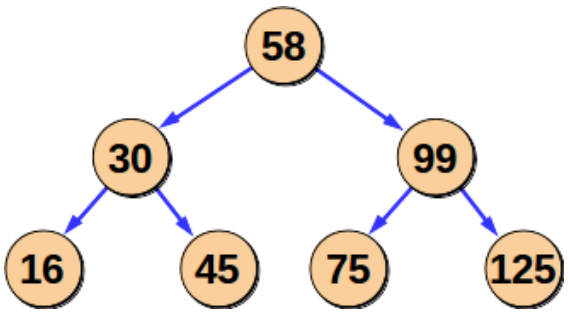
```
bt = BinaryTree()
bt.insert(50)
bt.insert(30)
bt.insert(20)
bt.insert(40)
bt.insert(70)
bt.insert(60)
bt.insert(80)

Egy adott érték keresése ( k = 40)
result = bt.search(40)
if result:
    print(f"Az {result.val} kulcs megtalálható a fában.")
else:
    print("A megadott kulcs nem található meg a fában.")
```

Program magyarázata.

1. TreeNode és BinaryTree osztályok: Ezek az osztályok ugyanazok, mint a előző példában. A `TreeNode` osztály reprezentálja a fa egy csomópontját, míg a `BinaryTree` osztály tartalmazza a fa gyökérelemét és a műveleteket, mint a beszúrást (`insert`) és a keresést (`search`).
2. Keresés (`search` és `_search_recursive` metódusok): A `search` metódus megkeresi az adott kulcsot (`key`) a bináris fában. A `_search_recursive` metódus rekurzívan keresi meg az adott kulcsot a fában, indulva a gyökérelem (`root`) alól.
3. Példa: Létrehozunk egy bináris fát (`bt`), beszúrunk néhány elemet, majd keresünk egy konkrét értéket (`key = 40`). Az eredményt megvizsgáljuk, és ha találjuk a kulcsot, akkor kiírjuk, hogy megtalálható a fában. Ez a példa bemutatja, hogyan lehet keresni egy adott kulcsot egy bináris fában, Pythonban. A `search` metódus visszaadja a talált csomópontot, ha megtalálja a kulcsot, vagy `None`-t, ha nem találja meg.

Hasonlítsuk össze az n elemű **tömbben** végzett keresést a **bináris fában** való kereséssel:

Keresés a tömbben: mindegyik összehasonlításkor: 1 elem elvetése. Összehasonlítások száma – N.



Mindegyik összehasonlításkor elveti a fennmaradó elemek felét. Összehasonlítások száma **~ log2N.**

Keresés bináris fában. Pascal.

Hozzunk létre egy keresőfüggvényt. A fő program függvénybemenetére két paraméter kerül: fa - a fa gyökerének címe és x - a kívánt szám. A függvény a csomópont címét adja vissza a keresett értékkel vagy nullával, ha nem található semmi.

```
function SearchInTree(Tree: PNode; x: integer): PNode;
begin
  if Tree = nil then begin
    Result := nil;
    Exit;
  end;
  if x = Tree^.data then
     Result := Tree
  else
  if x < Tree^.data then
       Result := SearchInTree(Tree^.left, x)
  else Result := SearchInTree(Tree^.right, x);
end;
```

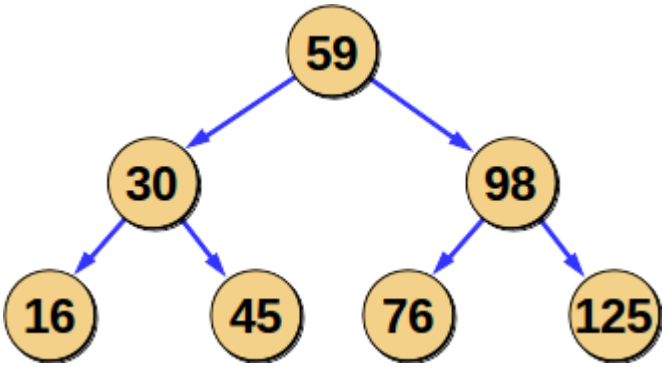
Keresési fa létrehozása.

A fa létrehozásához két paraméteres eljárást fogunk használni: Tree a fa gyökerének címe, x a hozzáadandó elem.

```
procedure AddToTree( var Tree: PNode; x: integer );
begin
  if Tree = nil then begin
    New(Tree);
    Tree^.data := x;
    Tree^.left := nil;
    Tree^.right := nil;
    Exit;
  end;
  if x < Tree^.data then
       AddToTree(Tree^.left, x)
  else AddToTree(Tree^.right, x);
end;
```

Fa bejárása

Ez a művelet az összes csomópont meghatározott sorrendben történő felsorolásához szükséges. Nézzük a megoldási lehetőségeket:



- bejárás bal – gyökér – jobb: 16, 30, 45, 59, 76, 98, 125;
- bejárás jobb – gyökér – bal: 125, 98, 76, 59, 45, 30, 16;
- bejárás gyökér – bal – jobb: 59, 30, 16, 45, 98, 76, 125;
- bejárás bal - jobb - gyökér: 16, 45, 30, 76, 125, 98, 59.

A bináris fa bejárása során a *bal-gyökér-jobb sorrendet* in-order bejárásnak nevezik. Ez a bejárás egyik klasszikus módszere, amely során a fa összes csomópontját rendezetten látogatjuk meg: először a bal oldali részfát, aztán a gyökérelemet, végül pedig a jobb oldali részfát.

Fa bejárásának algoritmusa. Python

```
#1 Fa bejárása
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert_recursive(self.root, key)
```

```
#2 Fa bejárása
    def _insert_recursive(self, root, key):
        if key < root.val:
            if root.left is None:
                root.left = TreeNode(key)
            else:
                self._insert_recursive(root.left, key)
        else:
            if root.right is None:
                root.right = TreeNode(key)
            else:
                self._insert_recursive(root.right, key)

    def inorder_traversal(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, root, result):
        if root:
            self._inorder_recursive(root.left, result)
            result.append(root.val)
            self._inorder_recursive(root.right, result)
```

Példa használat:

```
bt = BinaryTree()

bt.insert(50)

bt.insert(30)

bt.insert(20)

bt.insert(40)

bt.insert(70)

bt.insert(60)

bt.insert(80)

inorder_result = bt.inorder_traversal()

print("In-order bejárás eredménye:", inorder_result)  #
Output: In-order bejárás eredménye: [20, 30, 40, 50, 60, 70,
80]
```

Algoritmus magyarázata

1. TreeNode és BinaryTree osztályok: Ezek az osztályok ugyanazok, mint a korábbi példákban. A `TreeNode` osztály reprezentálja a fa egy csomópontját, míg a `BinaryTree` osztály tartalmazza a fa gyökérelemét és a műveleteket, mint a beszúrást (`insert`) és a bejárást (`inorder_traversal`).

2. In-order bejárás (`inorder_traversal` és `_inorder_recursive` metódusok): Az `inorder_traversal` metódus elindítja az in-order bejárást a fa gyökéreleméből. A `_inorder_recursive` metódus rekurzívan bejárja a fát in-order sorrendben, hozzáadva minden csomópont értékét a `result` listához.

3. Példa: Létrehozunk egy bináris fát (`bt`), beszúrunk néhány elemet, majd elvégezzük az in-order bejárást (`inorder_traversal`). Az eredményt kiírjuk, ami a fában lévő összes elemet rendezetten tartalmazza.

Az in-order bejárás az egyik leggyakrabban használt módszer bináris fákban, mert segít rendezett sorrendben feldolgozni a fa összes csomópontját.

Fa bejárásának algoritmusai. Pascal

A fa bejárása (tree traversal) algoritmusok Pascal nyelven is hasonló módon implementálhatóak, mint más nyelvekben. Az alábbiakban bemutatunk néhány alapvető fa bejárását Pascalban, különös figyelmet fordítva a mélységi bejárásra (depth-first traversal).

Mélységi bejárás (Depth-First Traversal)

A mélységi bejárás során a fát egy adott csomópontból kiindulva a gyermekcsomópontokat a lehető legmélyebben vizsgáljuk, mielőtt a testvér csomópontokat bejárnánk.

In-order bejárás (bal-gyökér-jobb)

Az in-order bejárás során a fa csomópontjait a következő sorrendben járjuk be: bal gyermek, gyökér, jobb gyermek.

```
type
  PTreeNode = ^TreeNode;
  TreeNode = record
    val: Integer;
    left, right: PTreeNode;
  end;

procedure InOrderTraversal(root: PTreeNode);
begin
  if root <> nil then
  begin
    InOrderTraversal(root^.left);
    Write(root^.val, ' ');
    InOrderTraversal(root^.right);
  end;
end;
```

Fő program

```
var
  root, node1, node2, node3: PTreeNode;
begin
  New(root); // Fa építése
  New(node1);
  New(node2);
  New(node3);

  root^.val := 1;
  node1^.val := 2;
  node2^.val := 3;
  node3^.val := 4;

  root^.left := node1;
  root^.right := node2;
  node1^.left := nil;
  node1^.right := nil;
  node2^.left := node3;
  node2^.right := nil;
  node3^.left := nil;
  node3^.right := nil;
  // In-order bejárás
  Write('In-order bejárás: ');
  InOrderTraversal(root);
  WriteLn;
end.
```

Post-order bejárás (bal-jobb-gyökér)

A post-order bejárás során a fa csomópontjait a következő sorrendben járjuk be: bal gyermek, jobb gyermek, gyökér.

```
procedure PostOrderTraversal(root: PTreeNode);
begin
  if root <> nil then
  begin
    PostOrderTraversal(root^.left);
    PostOrderTraversal(root^.right);
    Write(root^.val, ' ');
  end;
end;
```

Példa használat:

```
begin
  // ... fa építése (mint az előző példában)

  // Post-order bejárás
  Write('Post-order bejárás: ');
  PostOrderTraversal(root);
  WriteLn;
end.
```

Pre-order bejárás (gyökér-bal-jobb)

A pre-order bejárás során a fa csomópontjait a következő sorrendben járjuk be: gyökér, bal gyermek, jobb gyermek.

```
procedure PreOrderTraversal(root: PTreeNode);
begin
  if root <> nil then
  begin
    Write(root^.val, ' ');
    PreOrderTraversal(root^.left);
    PreOrderTraversal(root^.right);
  end;
end;
```

Példa használat:

```
begin
  // ... fa építése (mint az előző példában)
  Write('Pre-order bejárás: ');
  PreOrderTraversal(root);
  WriteLn;
end.
```

Program magyarázata

1. TreeNode rekord és típus deklaráció. Definiáljuk a `TreeNode` rekordot, amely tartalmazza a fa csomópontjait (`val` érték és `left`, `right` mutatók).
2. In-order, Post-order, Pre-order bejárások. Mindegyik bejárás egy rekurzív algoritmust használ. A `InOrderTraversal`, `PostOrderTraversal` és `PreOrderTraversal` függvények rekurzívan bejárják a fát a megfelelő sorrendben, és kiírják a csomópontok értékét.
3. Példa. Létrehozzuk a fát (`root` és gyerekei), majd elvégezzük az in-order, post-order és pre-order bejárásokat a `root` csomópontból kiindulva.

Ezek az algoritmusok segítenek megérteni és megvalósítani a bináris fa bejárását Pascal nyelven. A rekurzív megközelítés a leggyakoribb és legtisztább módszer a fa bejárásának

implementálására, mivel jól tükrözi a fa struktúráját és természetét.

Feladatok

Keresse meg egy bináris fa magasságát;

Keresse meg a k-ik maximális értéket a bináris fában;

Keresse meg a csomópontokat, amelyeknek a gyökértől való távolsága egyenlő k;

Keresse meg egy adott csomópont őseit egy bináris fában.

Vissza

IRODALOMJEGYZÉK

1. https://mek.oszk.hu/08400/08435/08435.pdf, Gérard Swinner,Tanuljunk meg programozni Python nyelven.
2. https://szit.hu/doku.php?id=oktatas:programozas:elemi_adatszerkezetek
3. https://oszkdk.oszk.hu/storage/00/00/59/28/dd/1/Adonyi_Adatstrukturak.pdf
4. http://www.informatom.hu/sze/01/LGB_SZ001/Algoritmusok_%C3%A9s_adatstrukt%C3%BAr%C3%A1k.pdf
5. https://web.uni-miskolc.hu/~matip/algoritmusok/
6. https://aries.ektf.hu/~hz/pdf-tamop/pdf-xx/algo-adatmodell.pdf
7. https://www.mechmat.univ.kiev.ua/wp-content/uploads/2021/09/pidruchnyk-alhorytmy-i-struktury-danykh.pdf

Алгоритми та структури даних/ Algoritmusok és adatszerkezetek методичні вказівки до практичних занять з дисципліни «Алгоритми та структури даних » для здобувачів першого (бакалаврського) рівня вищої освіти денної та заочної форм навчання, освітня програма: «Середня освіта (Інформатика)», галузь знань: «01 Освіта/Педагогіка», спеціальність (спеціалізація): «014 Середня освіта (014.04 Інформатика)» / Розробник: Йожеф Головач. – Берегове: ЗУІ ім. Ф.Ракоці ІІ, 2024, 48 с. (угорською мовою).

Метою методичного посібника є практичне використання структур даних для типових алгоритмах обробки даних, які вивчаються в курсі «Алгоритми та структури даних». У роботі описані основні структури даних, їх властивості, а також типові задачі, для яких вони використовуються. Наведені програми розв'язування задач на мові програмування Python, а деякі задачі – на мові Pascal. Методичні вказівки можуть бути використані на практичних заняттях та при самостійній роботі по даному курсу.

Виробничо-практичне видання

Алгоритми та структури даних/ Algoritmusok és adatszerkezetek

Методичні вказівки до практичних занять з дісципліни

«Алгоритми та структури даних»

2024 р.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 1 від13 серпня 2024 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол №22 від «26» серпня 2024 року)

Рекомендовано до видання в електронній формі (PDF) рішенням
Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол №7 від « 27» серпня 2024 року)

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ

Розробник методичних вказівок:

Йожеф ГОЛОВАЧ – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор технічних наук

Рецензенти:

Олександр МІЦА – завідувач кафедри інформаційних управляючих систем та технологій УжНУ, доктор техніічих наук, професор (м. Ужгород)

Мирослав СТОЙКА – доцент кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, кандидат фізико-математичних наук, доцент

Відповідальні за випуск:

Каталін КУЧІНКА – завідувач кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доцент, кандидат фізико-математичних наук

Олександр ДОБОШ – начальник Видавничого відділу ЗУІ ім. Ф.Ракоці ІІ

За зміст методичних вказівок відповідальність несе розробник.

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua) *Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції Серія ДК 7637 від 19 липня 2022 року*

Шрифт «Courier New». Розмір сторінок методичних вказівок: А4 (210х297мм).