Parcours en NSI

Ronan Charpentier

IREM Caen

16 décembre

Définition

Parcourir une structure de données signifie considérer successivement chacun de ses éléments, pour y appliquer éventuellement un traitement.

Structures

Les structures considérées ici sont les listes, les dictionnaires, les listes de dictionnaires (tables), les fichiers, les chaînes de caractères, les arbres et les graphes.

1ère Types construits

Itérer sur les éléments d'une liste
Itérer sur les éléments d'un dictionnaire

1ère Données en table

Rechercher les lignes d'une table vérifiant des critères

1ère Algorithmique

Parcours séquentiel d'un tableau, recherche occurrence, extremum, somme, *Tris, KNN, gloutons*

Tale Algorithmique

Parcourir un arbre de différentes façons (infixe, préfixe ou suffixe ; en largeur), Parcourir un graphe en profondeur d'abord, en largeur d'abord, Recherche textuelle

Liste : par index, par élément, avec enumerate

```
liste=[76, 14, 50, 35, 22, 29, 56, 44]
```

- `for i in range(len(liste)):`
- `for e in liste:`
- `for i,e in enumerate(liste):`

Dictionnaire : par clé, par valeur, par item

```
dict={'rep':42, 's':5050, 'pi':3}
```

- `for k in dico.keys():`
- `for v in dico.values():`
- `for k,v in dico.items():`

Occurrence d'un élément dans une liste

```
def rechercheLineaire(element,liste):
    for k in liste:
        if k==element:
            return True
    return False
```

Accumulation des éléments d'une liste

```
def accumulation(liste):
    somme=0
    for k in liste:
        somme=somme+k
    return somme
```

Recherche du minimum

```
def minimum(liste):
    m=liste[0]
    for e in liste[1:]:
        if e<m:
            m=e
    return m
```

Recherche de l'index du minimum

```
def indice(liste):
    i,m=0,liste[0]
    for j,e in enumerate(liste[1:]):
        if e<m:
        i,m=j,e
    return i
```

Tri sélection

Parcourir la liste pour trouver l'index du minimum. Echanger avec le premier élément. Recommencer avec le reste de la liste.

Tri insertion

Pour chaque élément de la liste, l'insérer dans le début de la liste en l'échangeant avec le précédent tant que celui-ci est plus grand.

Tri sélection

```
def triSelection(l):
    for i in range(len(l)):
        j,m=i,l[i]
        for k in range(j+1,len(l)):
            if l[k]<m:
                j,m=k,l[k]
        l[i],l[j]=l[j],l[i]
```

Parcourir la liste pour trouver l'index du minimum. Echanger avec le premier élément. Recommencer avec le reste de la liste.

https://urlz.fr/eqBf

Tri insertion

Pour chaque élément de la liste, l'insérer dans le début de la liste en l'échangeant avec le précédent tant que celui-ci est plus grand.

Tri sélection

Parcourir la liste pour trouver l'index du minimum. Echanger avec le premier élément. Recommencer avec le reste de la liste.

Tri insertion

```
def triInsertion(l):
    for i in range(len(l)):
        j=i
        while j>0 and l[j-1]>l[j]:
            l[j-1],l[j]=l[j],l[j-1]
            j=j-1
```

Pour chaque élément de la liste, l'insérer dans le début de la liste en l'échangeant avec le précédent tant que celui-ci est plus grand.

https://urlz.fr/eqBm

Première : recherche dichotomique dans une liste triée

```
def rechercheDicho(elt,liste):
    if liste==[]:
        return False
    m=len(liste)//2
    if elt==liste[m]:
        return True
    elif elt<liste[m]:
        return rechercheDicho(elt,liste[:m])
    else:
        return rechercheDicho(elt,liste[m+1:])
```

Terminale : recherche dans un arbre binaire de recherche

Première : recherche dichotomique dans une liste triée

Terminale : recherche dans un arbre binaire de recherche

```
def rechercheABR(elt,arbre):
    if arbre.estVide():
        return False
    r=arbre.racine()
    if elt==r:
        return True
    elif elt<r:
        return rechercheABR(elt,arbre.gauche())
    else:
        return rechercheABR(elt,arbre.droite())
```

Pile
+Pile() +estVide() : bool +empile(elt) : None +depile() : elt

File
+File() +estVide() : bool +enfile(elt) : None +defile() : elt

ArbreBinaire
+ArbreBinaire(r,g,d) +estVide() : bool +racine() : elt +gauche() : arbre +droite() : arbre

Graphe
+Graphe() +sommets() : list +voisins(s) : list +ajouteSommet(elt) : None +ajouteArete(s1,s2) : None

La structure de données adaptée est la **pile**.
Cette pile peut être explicite comme ci-dessous, ou implicite si on programme DFS de façon récursive.

```
def DFS(arbre):
    #la liste dejavus est initialement vide
    #on crée une pile vide
    #on empile la racine
    #tant que la pile n'est pas vide
        #dépiler le sommet et l'ajouter à dejavus
        #empiler les éventuels enfants du sommet
    #renvoyer dejavus
```

La structure de données adaptée est la **pile**.

```
def DFS(arbre):
    dejavus=[]
    p=Pile()
    p.empile(arbre.racine())
    while not p.estVide():
        s=p.depile()
        dejavus.append(s)
        if not arbre.gauche.estVide():
            p.empile(arbre.gauche.racine())
        if not arbre.droite.estVide():
            p.empile(arbre.droite.racine())
    return dejavus
```

La structure de données adaptée est la **file**.

```
def BFS(arbre):
    #la liste dejavus est initialement vide
    #on crée une file vide
    #on enfile la racine
    #tant que la file n'est pas vide
        #défiler le sommet et l'ajouter à dejavus
        #enfiler les éventuels enfants du sommet
    #renvoyer dejavus
```

La structure de données adaptée est la **file**.

```
def BFS(arbre):
    dejavus=[]
    f=File()
    f.enfile(arbre.racine())
    while not f.estVide():
        s=f.defile()
        dejavus.append(s)
        if not arbre.gauche.estVide():
            f.enfile(arbre.gauche.racine())
        if not arbre.droite.estVide():
            f.enfile(arbre.droite.racine())
    return dejavus
```

La seule modification par rapport à un BFS sur un arbre est qu'on n'enfile que les voisins qui n'ont pas encore été visités.

```
def BFS(graphe,sommet):
    dejavus=[]
    f=File()
    f.enfile(sommet)
    while not f.estVide():
        s=f.defile()
        dejavus.append(s)
        for v in graphe.voisins(sommet):
            if v not in dejavus:
                f.enfile(v)
         return dejavus
```

On n'empile les voisins d'un sommet qu'on vient d'explorer que si ces sommets n'ont pas encore été visités.

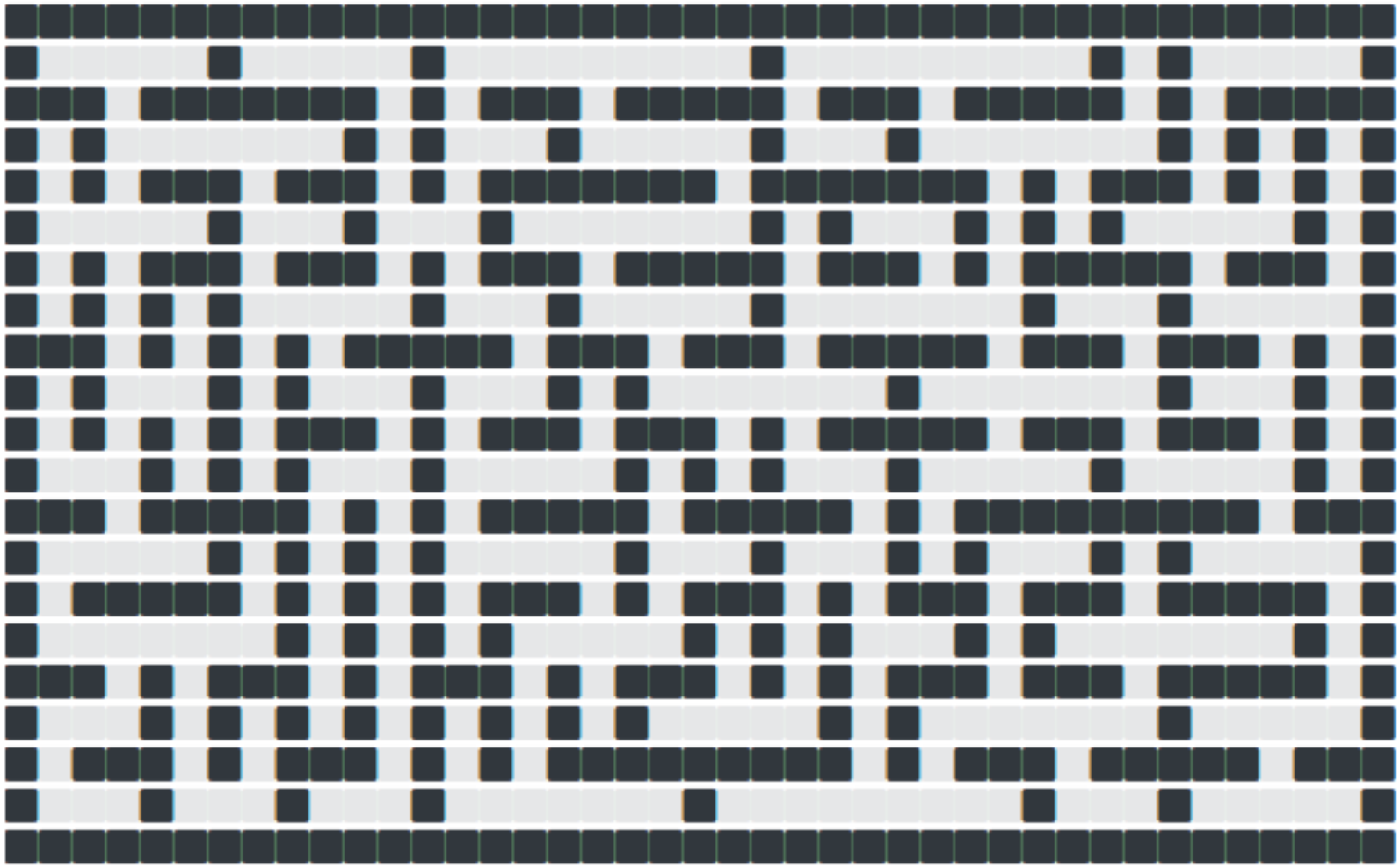
```
def DFS(graphe,sommet):
    dejavus=[]
    p=Pile()
    p.empile(sommet)
    while not p.estVide():
        s=p.depile()
        dejavus.append(s)
        for v in graphe.voisins(sommet):
            if v not in dejavus:
                p.empile(v)
    return dejavus
```

Au programme

- KNN k plus proches voisins
- Recherche de cycle dans un graphe
- Recherche de chemin dans un graphe
- Alignement de séquence
- Boyer-Moore

Hors-programme

- Coloriage glouton d'un graphe (Welsh-Powell)
- Arbre de couverture minimal d'un graphe (Prim, Kruskal)
- Chemin critique dans un graphe de précédence
- Composantes connexes d'un graphe
- Problème des huit reines de Gauss `https://urlz.fr/eqCt`
- Sudoku, séparation-évaluation, A*



Labyrinthes