Université Mohammed VI Polytechnique

Structures de données

Module : Structures de Données

M'hammed El Kahoui

April 19, 2022

Plan

Plan

Algorithmes et structures de données

```
Algorithms + Data structures = Programs
(Niklaus Wirth)
```

Algorithmes et structures de données

- La plupart des bons programmes fonctionnent grâce à des algorithmes efficaces et au choix de structures de données adéquates.

Definition

Une structure de données est une manière concrète d'organiser les données pour faciliter leur stockage, leur utilisation ainsi que leur modification.

Algorithmes et structures de données

- La plupart des bons programmes fonctionnent grâce à des algorithmes efficaces et au choix de structures de données adéquates.

Definition

Une structure de données est une manière concrète d'organiser les données pour faciliter leur stockage, leur utilisation ainsi que leur modification.

- Il n'a y a aucune structure de données qui réponde à tous les besoins.
- Il est donc important de connaître les forces et les limitations de chaque structure de données.

Example

Voici quelques exemples de structures de données

- Les structures indexées : l'exemple le plus simple est le tableau qui est bien adapté pour manipuler des données de même nature dont on connait le nombre à l'avance.
- Les structures récursives : on y trouve la liste qui est bien adaptée pour manipuler des données dont le nombre peut varier au cours de l'exécution d'un programme.

Types de données absraits

- Il est très utile de dissocier une structure de données de la manière dont elle a été réalisée.

Definition

Un type de donnéees abstrait (TDA) est la description d'un ensemble organisé de données et des opérations possibles sur cet ensemble de données.
C'est en quelque sorte une structure algébrique formée d'un ou plusieurs ensembles munis d'opérations qui vérifient des axiomes.

- Type abstrait de données = cahier de charge.
- Structure de données = réalisation du cahier de charges.

Interface d'une structure de données

- Toute structure de données doit avoir une interface.

Interface d'une structure de données

- Toute structure de données doit avoir une interface.
- L'interface est un ensemble de fonctions qui permettent de réaliser les opérations de base sur la structure de données.

Interface d'une structure de données

- Toute structure de données doit avoir une interface.
- L'interface est un ensemble de fonctions qui permettent de réaliser les opérations de base sur la structure de données. Par exemple
  - Entrées/Sorties : Saisie et affichage.
  - Accès aux différents constituants de la structure de données.
  - Recherche/Ajout/Suppression d'un élément dans la structure de données.

Plan

Listes simplement chaînées

- Liste : séquence ordonnée de données de même type.
- L'ordre compte : $(1, 2, 3, 4) \neq (2, 1, 3, 4)$.
- La multiplicité compte : $(1, 1, 2, 3, 4) \neq (1, 2, 3, 4)$.

Listes simplement chaînées

- Liste : séquence ordonnée de données de même type.
- L'ordre compte : $(1, 2, 3, 4) \neq (2, 1, 3, 4)$.
- La multiplicité compte : $(1, 1, 2, 3, 4) \neq (1, 2, 3, 4)$.
- Il est possible de représenter une liste par un tableau ou un pointeur (si le nombre d'éléments est une variable du programme).

Listes simplement chaînées

- Liste : séquence ordonnée de données de même type.
- L'ordre compte : $(1, 2, 3, 4) \neq (2, 1, 3, 4)$.
- La multiplicité compte : $(1, 1, 2, 3, 4) \neq (1, 2, 3, 4)$.
- Il est possible de représenter une liste par un tableau ou un pointeur (si le nombre d'éléments est une variable du programme).
- Ces deux représentations, dites contigues, exigent que la taille de la liste, appelée longueur, soit connue à l'avance.

Listes simplement chaînées

- Liste : séquence ordonnée de données de même type.
- L'ordre compte : $(1, 2, 3, 4) \neq (2, 1, 3, 4)$.
- La multiplicité compte : $(1, 1, 2, 3, 4) \neq (1, 2, 3, 4)$.
- Il est possible de représenter une liste par un tableau ou un pointeur (si le nombre d'éléments est une variable du programme).
- Ces deux représentations, dites contigues, exigent que la taille de la liste, appelée longueur, soit connue à l'avance.
- Il arrive souvent que les listes qu'on manipule se présentent de manière récursive, en particulier on ne connait pas à l'avance leur longueur.

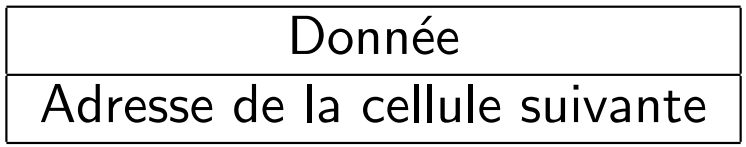
Definition

Une liste simplement chaînée est une liste d'objets, appelés maillons ou cellules, dans laquelle chaque cellule contient :

- une donnée qu'on veut stocker ou manipuler,
- l'adresse de la cellule suivante, ou une marque de fin s'il n'y a pas de cellule suivante.

- Chaque cellule dans une liste simplement chaînée est une structure qui contient deux champs :
  - le premier contient la donnée à stocker,
  - le second contient l'adresse de la cellule suivante.

- On peut donc représenter graphiquement une cellule comme suit :



- On peut aussi repésenter graphiquement une liste simplement chaînée comme suit :

Début et fin d'une liste simplement chaînée

- Deux positions sont très importantes dans une liste chaînée : le début, appelé tête, et la fin appelée queue.
- La tête d'une liste simplement chaînée est l'adresse de la première cellule de la liste.
- Sans la tête de la liste il est impossible de savoir où la liste commence.

Début et fin d'une liste simplement chaînée

- Deux positions sont très importantes dans une liste chaînée : le début, appelé tête, et la fin appelée queue.
- La tête d'une liste simplement chaînée est l'adresse de la première cellule de la liste.
- Sans la tête de la liste il est impossible de savoir où la liste commence.
- Étant donnée une cellule C dans une liste simplement chaînée, pour savoir où se trouve la cellule suivante il suffit de lire l'adresse écrite dans le deuxième champ de C.
- Ainsi, si l'adresse lue n'est pas valide (NULL) cela veut dire qu'on est à la fin de la liste.

Implémentation d'une liste simplement chaînée

Voici une manière d'implémenter en langage C une liste simplement chaînée de données de type int.

```
struct cellule
{
    int element;
    struct cellule * suivant;
};

typedef struct cellule cellule ;
typedef cellule * liste;
```

- L'instruction

  `cellule C;`

  vaut une déclaration d'un objet d'identificateur `C` et de type `cellule`.

- L'instruction

  `cellule C;`

  vaut une déclaration d'un objet d'identificateur `C` et de type `cellule`.
- L'instruction

  `Liste L;`

  vaut une déclaration d'un objet d'identificateur `L` et de type pointeur vers une `cellule`.

Interface basique d'une liste chaînée

Nous décrivons dans la suite quelques opérations fondamentales sur les listes chaînées.

Test à vide	:	Savoir si une liste chaînée est vide.
Longueur d'une liste	:	Calculer le nombre d'éléments d'une liste.
Ajout d'une cellule	:	Ajouter une cellule à une liste.
Affichage	:	Affichage des éléments d'une liste.
Suppression	:	Supprimer une cellule sans récupérer la donnée.
Extraction	:	Récupérer la donnée d'une cellule et supprimer la cellule.
Destruction d'une liste	:	Libérer les zone-mémoires allouées aux cellules.

Test à vide d'une liste chaînée

```
int testVide(liste L)
{
    if(L == NULL)
        return 1;
    return 0;
}
```

Longueur d'une liste

```
unsigned longueur(liste L)
{
    unsigned l=0;
    if(L != NULL)
    {
        liste temp=L;
        while(temp != NULL)
        {
            l=l+1;
            temp=temp->suivant;
        }

    }
    return l;
}
```

Ajout d'un élément au début d'une liste

```
void ajoutDebut(int x, liste * L)
{
    // Déclaration de la cellule à ajouter.
    liste C = (cellule *)malloc(sizeof(cellule));
    // Initialisation de la cellule à ajouter.
    C->element = x;
    C->suivant = *L;
    // Chainage de C au début de L.
    *L = C;
    return;
}
```

Ajout d'un élément à la fin d'une liste

```
void ajoutFin(int x, liste * L)
{
    // Déclaration de la nouvelle cellule.
    liste C = (cellule *)malloc(sizeof(cellule));
    // Initialisation de la cellule à ajouter.
    C->element = x;
    C->suivant = NULL;
    if(*L==NULL)
        *L = C;
    else
    {
        Déplacement vers la fin de liste.
        liste temp = *L;
        while(temp->suivant != NULL)
            temp = temp->suivant;
        Chainage de C à la fin de L.
        temp->suivant = C;
    }
    return;
}
```

Affichage du contenu d'une liste chaînée

```
void affichageListe(liste L)
{
    liste temp=L;
    printf("[");
    while(temp->suivant!=NULL)
    {
        printf("%d ",temp->element);
        temp=temp->suivant;

    }
    printf("%d]",temp->element);
}
```

Suppression du premier élément d'une liste

```
void suppressionDebut(liste * L)
{
    if(*L != NULL)
    {
        liste temp = *L;
        // Déplacer la tête de liste vers la seconde cellule.

        *L = (*L)->suivant;
        // Libérer la mémoire allouée à la première cellule.
        free(temp);
    }
    return;
}
```

Suppression du dernier élément d'une liste chaînée

```
void suppressionFin(liste *L)
{
    if(*L==NULL)
        return;
    if((*L)->suivant == NULL)
    {
        free(*L);
        *L=NULL;
    }
    else
    {
        // Accéder à la dernière et avant dernière cellules.
        liste temp = *L;
        liste temp_1 = *L;
        while(temp->suivant != NULL)
        {
            temp_1 = temp;
            temp = temp->suivant;
        }
        // L'avant dernière cellule devient la dernière.
        temp_1->suivant = NULL;
        // Libérer la mémoire allouée à la dernière cellule.
        free(temp);
    }
    return;
}
```

Extraction du premier élément d'une liste chaînée

```
int extractionDebut(liste *L)
{
    int res = (*L)->element;
    suppressionDebut(L);
    return res;
}
```

Extraction du dernier élément d'une liste chaînée

```
int extractionFin(liste * L)
{
    int res;
    // Accéder à la dernière et avant dernière cellules.
    liste temp = *L;
    liste temp_1 = *L;
    while(temp->suivant != NULL)
    {
        temp_1 = temp;
        temp = temp->suivant;
    }
    // Extraction du dernier élément de la liste.
    res = temp->element;
    // L'avant dernière cellule devient la dernière.
    temp_1->suivant = NULL;
        // Libérer la mémoire allouée à la dernière cellule.
    free(temp);
    return res;
}
```

Destruction d'une liste

```
void liberationListe(liste L)
{
    while(L != NULL)
    {
        liste temp = L;
        L = L->suivant;
        free(temp);
    }
}
```

Plan

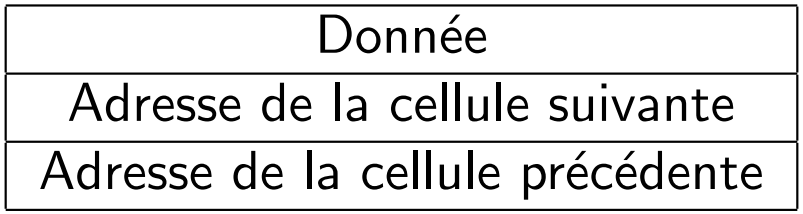
Listes doublement chaînées

Definition

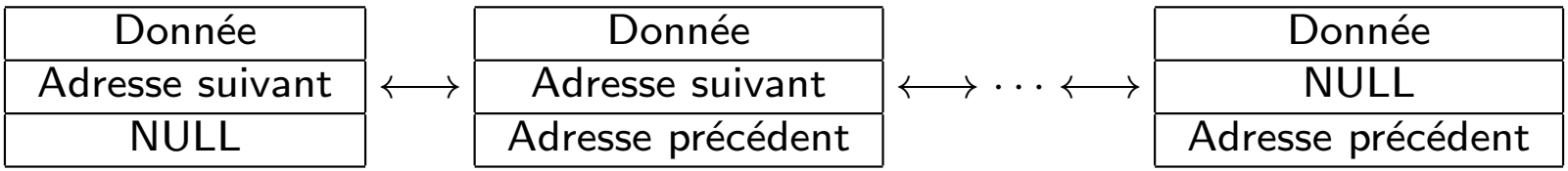
Une liste doublement chaînée est une liste d'objets, appelés maillons ou cellules, dans laquelle chaque cellule contient :

- la donnée qu'on veut stocker ou manipuler,
- l'adresse de la cellule suivante, ou une marque de fin s'il n'y a pas de cellule suivante,
- l'adresse de la cellule précédente, ou une marque de début s'il n'y a pas de cellule précédente.

On peut donc représenter graphiquement une cellule comme suit :



On peut aussi représenter graphiquement une liste doublement chaînée comme suit :



Début et fin d'une liste doublement chaînée

- Deux positions sont importantes dans une liste doublement chaînée : le début et la fin.

Début et fin d'une liste doublement chaînée

- Deux positions sont importantes dans une liste doublement chaînée : le début et la fin.
- Le début d'une liste doublement chaînée est l'adresse de la première cellule de la liste.
- Sans le début de la liste il est impossible de savoir où la liste commence.

Début et fin d'une liste doublement chaînée

- Deux positions sont importantes dans une liste doublement chaînée : le début et la fin.
- Le début d'une liste doublement chaînée est l'adresse de la première cellule de la liste.
- Sans le début de la liste il est impossible de savoir où la liste commence.
- Étant donnée une cellule C dans une liste doublement chaînée, pour savoir où se trouve la cellule suivante il suffit de lire l'adresse écrite dans le deuxième champ de C.
- Ainsi, si l'adresse lue n'est pas valide (NULL) cela veut dire qu'on est à la fin de la liste.

Début et fin d'une liste doublement chaînée

- Deux positions sont importantes dans une liste doublement chaînée : le début et la fin.
- Le début d'une liste doublement chaînée est l'adresse de la première cellule de la liste.
- Sans le début de la liste il est impossible de savoir où la liste commence.
- Étant donnée une cellule C dans une liste doublement chaînée, pour savoir où se trouve la cellule suivante il suffit de lire l'adresse écrite dans le deuxième champ de C.
- Ainsi, si l'adresse lue n'est pas valide (NULL) cela veut dire qu'on est à la fin de la liste.
- Étant donnée une cellule C dans une liste doublement chaînée, pour savoir où se trouve la cellule précédente il suffit de lire l'adresse écrite dans le troisième champ de C.
- Ainsi, si l'adresse lue n'est pas valide (NULL) cela veut dire qu'on est au bébut de la liste.

Implémentation d'une liste doublement chaînée

Voici une manière d'implémenter en C une liste doublement chaînée de données de type int.

```
struct celluleDouble
{
    int element;
    struct celluleDouble * suivant;
    struct celluleDouble * precedent;
};
typedef struct celluleDouble celluleDouble;
```

```c
struct listeDouble
{
    celluleDouble * debut;
    celluleDouble * fin;
    unsigned longueur;
};
typedef struct listeDouble listeDouble;
```

- Le champ longueur représente la longueur de la liste.
- Le champ debut représente l'adresse de la première cellule de la liste.
- Le champ fin représente l'adresse de la dernière cellule de la liste.

- L'instruction

  `celluleDouble C;`

  vaut une déclaration d'un objet d'identificateur `C` et de type `celluleDouble`.

- L'instruction

  `celluleDouble C;`

  vaut une déclaration d'un objet d'identificateur `C` et de type `celluleDouble`.
- L'instruction

  `listeDouble L;`

  vaut une déclaration d'un objet d'identificateur `L` et de type `listeDouble`.

Opérations fondamentales sur une liste doublement chaînée

Nous décrivons dans la suite quelques opérations fondamentales sur les listes doublement chaînées.

Test à vide	:	Savoir si une liste doublement chaînée est vide.
Longueur	:	Calculer le nombre d'éléments d'une liste.
Ajout d'une cellule	:	Ajouter une cellule à une liste.
Affichage	:	Affichage des éléments d'une liste.
Suppression	:	Supprimer une cellule sans récupérer la donnée.
Extraction	:	Récupérer la donnée d'une cellule et supprimer la cellule.
Destruction une liste	:	Libérer les zone-mémoires allouées aux cellules.

Test à vide

```
int testVide(listeDouble L)
{
    if(L.debut == NULL)
        return 1;
    else
        return 0;
}
```

Longueur

```
unsigned longueur(listeDouble L)
{
    if(testVide(L)==1)
        return 0;
    return L.longueur;
}
```

- Le fait que la fonction longueur est aussi simple est dû à l'ajout de la longueur comme champ dans la structure listeDouble.
- Mais il ne faut pas oublier de mettre à jour le champ longueur à chaque ajout ou suppression dans une liste.

Ajout d'un élément au début

```
void ajoutDebut(int x, listeDouble * L)
{
    // Déclaration et initialisation de la cellule à ajouter.
    celluleDouble * C = (celluleDouble *)malloc(sizeof(celluleDouble));
    C->element = x;
    C->suivant = L->debut;
    C->precedent=NULL;
    if(testVide(*L)==1)
        {
            L->debut=C;
            L->fin=C;
            L->longueur+=1;
            return;
        }
    // Double chainage de C au début de L et mise à jour de la longueur.
    L->debut->precedent=C;
    L->debut = C;
    L->longueur+=1;
    return;
}
```

Ajout d'un élément à la fin

```
void ajoutFin(int x, listeDouble *L)
{
    if(testVide(*L)==1)
        ajoutDebut(x,L);
    else
    {
        // Déclaration et initialisation de la nouvelle cellule.
        celluleDouble *C = (celluleDouble *)malloc(sizeof(celluleDouble));
        C->element = x;
        C->suivant = NULL;
        C->precedent=L->fin;
        L->fin->suivant=C;
        L->fin=C;
        L->longueur+=1;
    }
    return;
}
```

Affichage

```
void affichageListeDouble(listeDouble L)
{
    celluleDouble *temp=L.debut;
    printf("[");
    while(temp->suivant!=NULL)
    {
        printf("%d ",temp->element);
        temp=temp->suivant;

    }
    if(temp!=NULL)
        printf("%d",temp->element);
    printf("]");
}
```

Suppression du premier élément

```
void suppressionDebut(listeDouble *L)
{
    if(L->debut != NULL)
    {
        celluleDouble * temp = L->debut;
        // Déplacer la tête de liste vers la seconde cellule.

        L->debut = L->debut->suivant;
        L->debut->precedent=NULL;
        // Libérer la mémoire allouée à la première cellule.
        free(temp);
        L->longueur-=1;
    }
    return;
}
```

Suppression du dernier élément

```
void suppressionFin(listeDouble * L)
{
    if(L->debut!= NULL)
    {
        celluleDouble * temp=L->fin;
        L->fin=L->fin->precedent;
        L->fin->suivant=NULL;
        free(temp);
        L->longueur-=1;
    }
    return;
}
```

Extraction du premier élément

```
int extractionDebut(listeDouble * L)
{
    int res = L->debut->element;
    suppressionDebut(L);
    return res;
}
```

Extraction du dernier élément

```
int extractionFin(listeDouble *L)
{
    int res = L->fin->element;
    suppressionFin(L);
    return res;
}
```

Destruction

```
void liberer(listeDouble L)
{
    while(L.debut != NULL)
    {
        celluleDouble * temp = L.debut;
        L.debut = L.debut->suivant;
        free(temp);
    }
}
```

Plan

Généralités

- Les piles et les files sont des ensembles dynamiques pour lesquels la consultation, l'insertion et la suppression d'un élément se font toujours du même côté.

Généralités

- Les piles et les files sont des ensembles dynamiques pour lesquels la consultation, l'insertion et la suppression d'un élément se font toujours du même côté.
- La pile fonctionne selon le principe LIFO (last in first out) : l'élément supprimé est toujours le dernier à avoir été inséré.

Généralités

- Les piles et les files sont des ensembles dynamiques pour lesquels la consultation, l'insertion et la suppression d'un élément se font toujours du même côté.
- La pile fonctionne selon le principe LIFO (last in first out) : l'élément supprimé est toujours le dernier à avoir été inséré.
- La file fonctionne selon le principe FIFO (first in first out) : l'élément supprimé est toujours le plus ancien : le premier à avoir été inséré.

Généralités

- Les piles et les files sont des ensembles dynamiques pour lesquels la consultation, l'insertion et la suppression d'un élément se font toujours du même côté.
- La pile fonctionne selon le principe LIFO (last in first out) : l'élément supprimé est toujours le dernier à avoir été inséré.
- La file fonctionne selon le principe FIFO (first in first out) : l'élément supprimé est toujours le plus ancien : le premier à avoir été inséré.
- Les piles et les files sont utilisées pour le stockage temporaire des données.

Exemples d'utilisation

- Piles
  - Traitement de texte : Ctrl-Z.
  - Navigateurs : historique de navigation.
  - Programmes récursifs : Piles d'appels.
- Files
  - Gestion des requêtes d'impression dans une imprimante.
  - Répartition du temps-machine dans un système d'exploitation multitâches.

Piles

- Une pile est un ensemble dynamique ne permettant des insertions ou des suppressions qu'à une seule extrémité, appelée sommet de la pile.

Piles

- Une pile est un ensemble dynamique ne permettant des insertions ou des suppressions qu'à une seule extrémité, appelée sommet de la pile.
- Empiler un objet x dans une pile P consiste à insérer x au sommet de P.
- Dépiler une pile P consiste à supprimer de P l'objet placé au sommet et le retourner comme résultat du traitement.

Piles

- Une pile est un ensemble dynamique ne permettant des insertions ou des suppressions qu'à une seule extrémité, appelée sommet de la pile.
- Empiler un objet x dans une pile P consiste à insérer x au sommet de P.
- Dépiler une pile P consiste à supprimer de P l'objet placé au sommet et le retourner comme résultat du traitement.
- On ne peut accéder à un élement stocké dans une pile que lorsqu'on a dépilé tous les éléments stockés "au-dessus" de cet élément.

Spécification d'une pile

pileVide()	:	Teste si une pile est vide.
lire()	:	Permet de lire l'élément au sommet de la pile.
empiler()	:	Permet d'empiler un élément dans une pile.
depiler()	:	Permet de dépiler une pile.

- La fonction depiler() ne peut être appelée qu'avec une pile non vide.

Implémentation via une liste simplement chaînée

On se propose de réaliser une pile d'entiers à travers une liste chaînée.

```
struct cellule
{
    int element;
    struct cellule * suivant;
};
typedef struct cellule cellule ;
typedef cellule * pile;
```

Lecture du sommet de la pile

```
int lire(pile L)
{
    if(L != NULL)
        return L->element;
    else
        exit(EXIT_FAILURE);
}
```

Test à vide d'une pile

```
int pileVide(pile L)
{
    if(L == NULL)
        return 1;
    else
        return 0;
}
```

Empilage

```
void empiler(int x, pile *L)
{
    // Déclaration de la cellule à ajouter.
    pile C = (pile)malloc(sizeof(cellule));
    // Initialisation de la cellule à ajouter.
    C->element = x;
    C->suivant = *L;
    // Empilement.
    *L = C;
    return;
}
```

Dépilage

```
int depiler(pile * L)
{
    if(*L != NULL)
    {
        int res=(*L)->element;
        pile temp = *L;
        // Déplacer la tête de pile vers la seconde cellule.
        *L = (*L)->suivant;
        // Libérer la mémoire allouée à la première cellule.
        free(temp);
        return res;
    }
    else
        exit(EXIT_FAILURE);
}
```

Files

- Une file est un ensemble dynamique qui a un début et une fin.
- Enfiler un élément dans une file F consiste à l'ajouter à la fin de F.
- Défiler une file F consiste à supprimer l'élément qui se trouve au début de L et le retourner comme résultat du traitement.

Spécification d'une file

fileVide()	:	Teste si une file est vide.
lire()	:	Permet de lire l'élément au début de la file.
enfiler()	:	Permet d'enfiler un élément dans une file.
defiler()	:	Permet de défiler une file.

- La fonction defiler() ne peut être appelée qu'avec une file non vide.

Implémentation via une liste doublement chaînée

```
struct celluleDouble
{
    int element;
    struct celluleDouble * suivant;
    struct celluleDouble * precedent;
};
typedef struct celluleDouble celluleDouble;
```

```c
struct file
{
    celluleDouble * debut;
    celluleDouble * fin;
};
typedef struct file file;
```

Lecture du début de la file

```
int lire(file L)
{
    if(L.debut != NULL)
        return L.debut->element;
    else
        exit(EXIT_FAILURE);
}
```

Test à vide d'une file

```
int fileVide(file L)
{
    if(L.debut == NULL)
        return 1;
    else
        return 0;
}
```

Enfilage

```
void enfiler(int x, file * L)
{
    // Déclaration et initialisation de la nouvelle cellule.
    celluleDouble * C = (celluleDouble *)malloc(celluleDouble);
    C->element = x;
    C->suivant = NULL;
    if(fileVide(*L)==1)
    {
        C->precedent=NULL;
        L->debut=C;
        L->fin=C;
    }
    else
    {
        C->precedent=L->fin;
        L->fin->suivant=C;
        L->fin=C;
    }
    return;
}
```

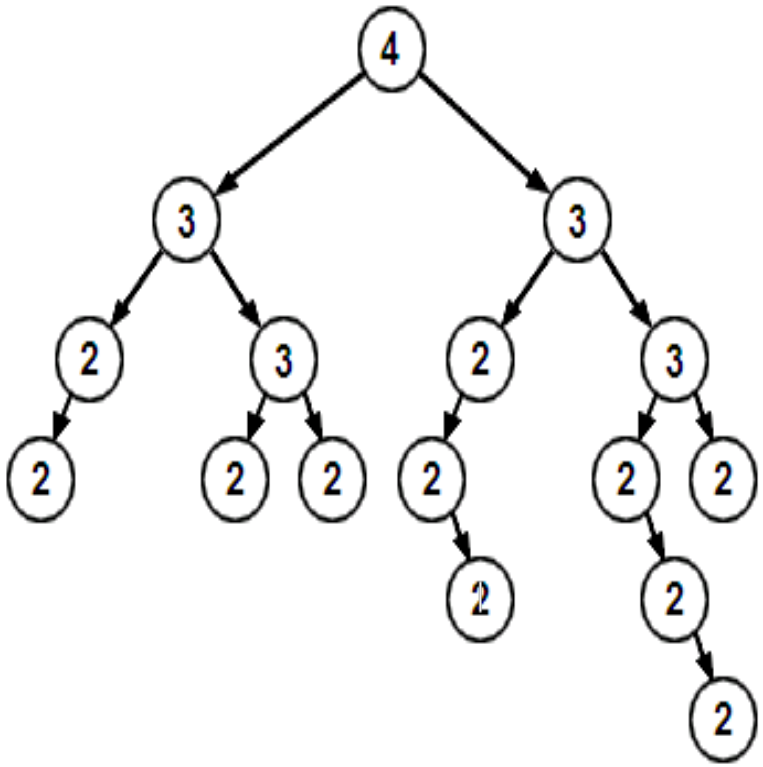
Défilage

```
int defiler(file * L)
{
    if(L->debut != NULL)
    {
        int res=L->debut->element;
        celluleDouble * temp=L->debut;
        L->debut=L->debut->suivant;
        free(temp);
        return res;
    }
    else
        exit(EXIT_FAILURE);
}
```

Plan

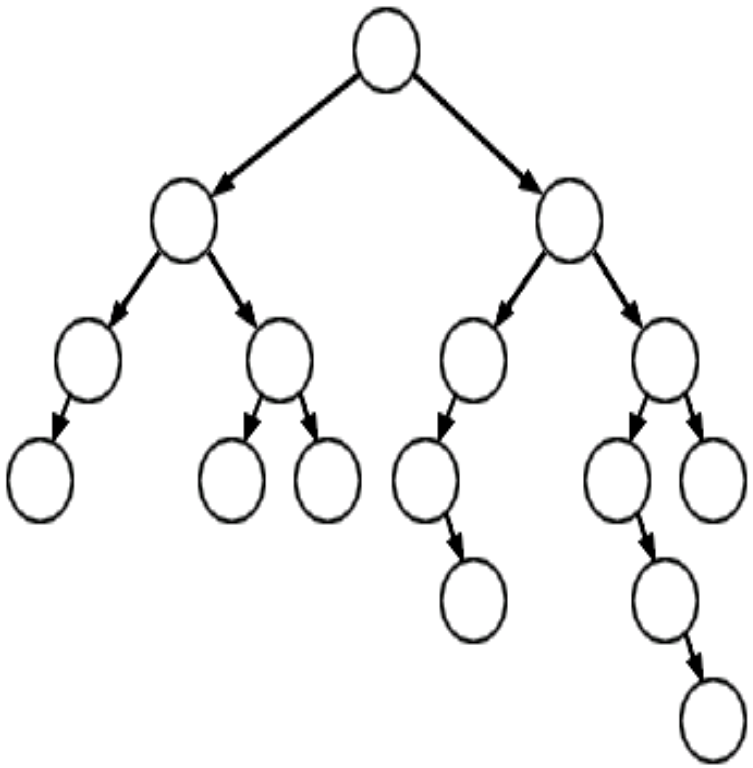
Commençons par un exemple

- Une représentation arborescente contient des données et des relations entre les données.



Commençons par un exemple

- Représentation des relations entre les données.



Arborescences

Definition

Une arborescence, ou arbre enraciné, est une structure de données contenant un nombre fini d'éléments, rangés dans des noeuds, telle que :

- lorsque la structure n'est pas vide, un noeud particulier appelé racine sert de point de départ,
- chaque noeud autre que la racine est attaché à un seul autre noeud, appelé son parent, mais la racine n'a pas de parent,
- chaque noeud possède zéro, un ou plusieurs noeuds qui lui sont attachés, appelés ses fils,
- tous les noeuds ont la racine comme ancêtre commun (parent, ou parent de parent, ...).

Mettons des mathématiques la-dessus

- $(E, \leq)$ ensemble ordonné :
  - Reflexivité : $\forall\, x \in E \quad x \leq x$.
  - Symétrie : $\forall\, x, y \in E \quad (x \leq y \text{ et } y \leq x) \implies x = y$.
  - Transitivité : $\forall\, x, y, z \in E \quad (x \leq y \text{ et } y \leq z) \implies x \leq z$.
  - $(E, \leq)$ totalement ordonné : $\forall\, x, y \in E \quad x \leq y \text{ ou } y \leq x$.

Mettons des mathématiques la-dessus

- $(E, \leq)$ ensemble ordonné :
  - Reflexivité : $\forall\, x \in E \quad x \leq x$.
  - Symétrie : $\forall\, x, y \in E \quad (x \leq y \text{ et } y \leq x) \implies x = y$.
  - Transitivité : $\forall\, x, y, z \in E \quad (x \leq y \text{ et } y \leq z) \implies x \leq z$.
  - $(E, \leq)$ totalement ordonné : $\forall\, x, y \in E \quad x \leq y \text{ ou } y \leq x$.
- Un arbre enraciné est $(E, \leq, r)$ tel que :
  - $(E, \leq)$ ensemble ordonné.
  - $\forall\, x \in E \quad r \leq x$.
  - $\forall\, x \in E$ l'ensemble

    $$E_x = \{y \in E \mid y \leq x\}$$

    est totalement ordonnée par $\leq$.

Vocabulaire sur les arbres

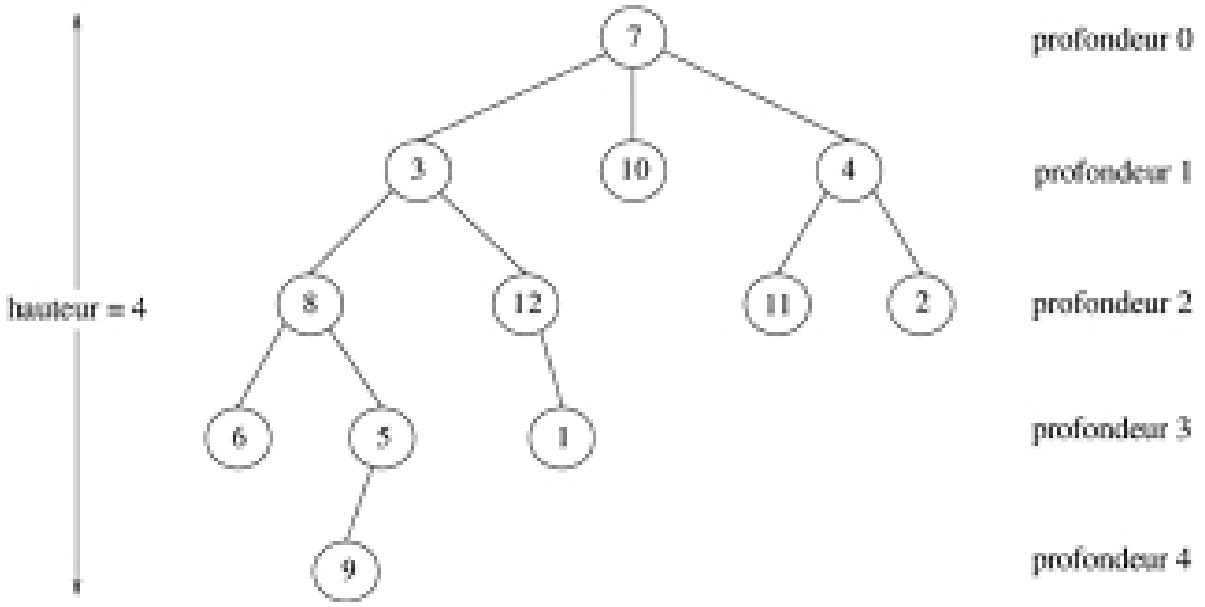
- Un arbre est dit binaire si chaque noeud possède au plus deux fils.
- Dans un arbre, un noeud qui n'a pas de fils est appelé une feuille.
- Un chemin dans un arbre est une suite finie $x_0, \ldots, x_n$ de noeuds telle que x_{i+1} est un fils de x_i pour chaque $i = 0, \ldots, n-1$.
- La longueur d'un chemin $x_0, \ldots, x_n$ est égale à n.
- Un noeud y de T est un descendant d'un noeud x s'il existe un chemin dans l'arbre T qui commence en x et termine en y.
- Un noeud x de T est un ancêtre d'un noeud y s'il existe un chemin dans l'arbre T qui commence en x et termine en y.

Hauteur et profondeur

- La hauteur d'un noeud x de T est la longueur du plus long chemin de T qui commence en x.
- La hauteur de l'arbre T est tout simplement la hauteur de sa racine.

Hauteur et profondeur

- La hauteur d'un noeud x de T est la longueur du plus long chemin de T qui commence en x.
- La hauteur de l'arbre T est tout simplement la hauteur de sa racine.
- La profondeur d'un noeud x de T est la longueur du chemin qui mène de la racine de T jusqu'à x.
  - La racine d'un arbre est de profondeur 0.
  - La profondeur d'un noeud est égale à la profondeur de son père plus 1.
  - Si un noeud est de profondeur p alors tous ses fils sont de profondeur $p + 1$.


hauteur = 4
7
3
10
4
8
12
11
2
6
5
1
9
profondeur 0
profondeur 1
profondeur 2
profondeur 3
profondeur 4

Arbres binaires

- La représentation la plus naturelle d'un arbre binaire est basée sur sa définition récursive :

Arbres binaires

- La représentation la plus naturelle d'un arbre binaire est basée sur sa définition récursive : un arbre binaire est soit l'arbre vide soit formé d'une racine et deux arbres binaires disjoints, appelés sous-arbre gauche et sous-arbre droit.
- Ainsi, chaque noeud dans un arbre est une structure qui contient un champ pour le contenu, ou la clé, du noeud et deux champs qui représentent respectivement le sous-arbre gauche et le sous-arbre droit descendant du noeud.

```c
struct noeud
{
    int cle;
    struct noeud * gauche;
    struct noeud * droit;

};
typedef struct noeud noeud;
typedef noeud * arbre;
```

Spécification de l'arbre binaire

Nous commençons par les opérations d'accès aux différents constituants d'un arbre.

EstVide(T)	:	Teste si un arbre binaire T est vide.
Racine(T)	:	Renvoie la racine d'un arbre T.
Gauche(T)	:	Renvoie le sous-arbre gauche d'un arbre T.
Droit(T)	:	Renvoie le sous-arbre droit d'un arbre T.

- La seconde opération est indéfinie pour un arbre vide.

Nous donnons maintenant les opérations basiques de construction et de modification d'un arbre

ArbreVide()	:	Crée un arbre binaire vide.
FaireArbre(x, g, d)	:	Crée un arbre dont la racine contient x et les sous-arbres gauche et droit sont g et d.
FixerCle(x,a)	:	Affecte la valeur x à la racine de l'arbre a.
FixerGauche(g,a)	:	Remplace le sous-arbre gauche de a par g.
FixerDroit(d,a)	:	Remplace le sous-arbre droit de a par d.

- Les trois dernières opérations ne sont pas définies pour un arbre vide.

Test à vide d'un arbre

```
int estVide(arbre T)
{
    if(T==NULL)
        return 1;
    return 0;
}
```

Extraction de la racine

```
int racine(arbre T)
{
    if(estVide(T))
        exit(EXIT_FAILURE);
    return T->cle;
}
```

Extraction du sous-arbre gauche

```
arbre gauche(arbre T)
{
    if(estVide(T))
        return NULL;
    return T->gauche;
}
```

Extraction du sous-arbre droit

```
arbre droit(arbre T)
{
    if(estVide(T))
        return NULL;
    return T->droit;
}
```

Construction des arbres

```
arbre faireArbre(int x,arbre T_1,arbre T_2)
{
    arbre T=(arbre)malloc(sizeof(noeud));
    T->cle=x;
    T->gauche=T_1;
    T->droit=T_2;
    return T;
}
```

Modification de la racine

```
void fixerCle(int x,arbre * T)
{
    if(estVide(T))
        exit(EXIT_FAILURE);
    else
    {
        (*T)->cle=x;
    }
}
```

Modification du sous-arbre gauche

```
void fixerGauche(arbre G, arbre * T)
{
    if(estVide(T))
        exit(EXIT_FAILURE);
    else
    {
        (*T)->gauche=G;
    }
}
```

Modification du sous-arbre droit

```
void fixerDroit(arbre D, arbre * T)
{
    if(estVide(T))
        exit(EXIT_FAILURE);
    else
    {
        (*T)->droit=D;
    }
}
```