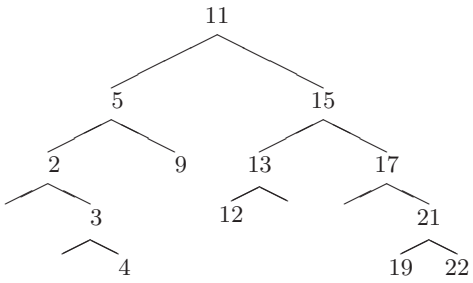
1. **Arbre binaire de recherche**. Rappel : un ABR est un arbre tel que la valeur de la racine est supérieure à celle de son fils gauche, et inférieure à celle de son fils droit (et dont tous les sous-arbres sont des ABR). Il n'y a donc qu'une seule possibilité pour construire l'ABR demandé, si on ajoute les valeurs dans l'ordre donné :

(a)



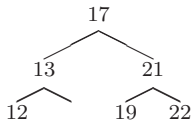
(b) Profondeur : 4 (longueur en nombre d'arcs de la plus longue branche)

(c) 11/15/17/21/19; 11/5/9.

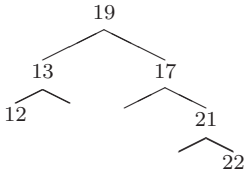
(d) La feuille la plus à gauche de son sous-arbre droit : ici, valeur 12. Ou la feuille la plus à droite du sous-arbre gauche, ici, 9. Non seulement on est sûr que sa valeur est comprise entre celles des fils gauches de la racine, mais en plus on est sûr, si on prend la valeur du "coin" (gauche ou droit), de ne pas avoir dans l'arbre une valeur mal placée (contrairement à ce qui se passerait si on prenait, par exemple, 13, qui est bien comprise en 5 et 15).

(e) Deux solutions : (1) on remonte le 17, ce qui conduit à la réorganisation (a) ; ou (2) on s'inspire du cas précédent : on remplace 15 par la feuille la plus à gauche de son sous-arbre droit (19) : (b)[3].

(a)



(b)



(f) Généralisation : on définit la notion de « coin gauche » (ou droit) : la feuille la plus à gauche d'un (sous-)arbre donné. Dans ce cas, l'algorithme peut être formulé ainsi : soit x le nœud à supprimer. Si x a un fils droit, remplacer x par le coin gauche de son fils droit (que l'on supprime alors), sinon, s'il a un fils gauche, le remplacer par le coin droit de son fils gauche, sinon (il n'a pas de fils) le supprimer tout simplement.

(g) Préfixe : 11 5 2 3 4 9 15 13 12 17 21 19 22
Postfixe : 4 3 2 9 5 12 13 19 22 21 17 15 11

(h) La version récursive est plus simple, mais on pouvait aussi proposer une version itérative. Le point clé est d'afficher le nœud après le traitement de son fils gauche, et avant le traitement de son fils droit (ou du passage au frère).

```
void parcours(noeud x)
{
  if (existe_fg(x)) parcours(fg(x)) ;
  afficher (x) ;
  if (existe_fd(x)) parcours(fd(x)) ;
}
```

[3]Pour des raisons de place, on ne représente ici que le fils droit de 11.

2. **Affixes**

Idée générale : les affixes peuvent être définis par un intervalle d'indices dans le tableau chaîne de caractères. Si n est le nombre de lettre du mot s, il faut produire deux séries de chaînes : celles dont les indices sont (1,1), (1,2), (1,3)... jusqu'à (1,$n-1$) ; et celles dont les indices sont (2,n), (3,n), (4,n)... jusqu'à (n,n). Cette idée est mise en œuvre dans le code suivant :

```
void affiche_sous_chaine(char s[], int d, int f)
{
  int i ;
  for (i=d ; i<=f ; i++)
    printf("%c", s[i]) ;
  printf("\n") ;
}

void aff_affixes(char s[])
{
  int l = strlen(s) ;
  int k ;

  // Deux boucles pour plus de lisibilité
  // D'abord préfixes
  for (k=1 ; k<=(l-1) ; k++) {
    affiche_sous_chaine(s, 0, k-1) ;
  }

  // ... ensuite suffixes
  for (k=1 ; k<=(l-1) ; k++) {
    affiche_sous_chaine(s, k, l) ;
  }
}

int main(int argc, char ** argv)
{
  if (argc <= 1) exit(0) ;
  aff_affixes(argv[1]) ;
}
```

Complexité : les appels à `affiche_sous_chaine()` sont fait $l-1$ fois dans chaque boucle (l étant la longueur de la chaîne). Cela fait donc $2 \times (l-1)$ appels, mais ces appels font des nombres de tours variables :

Le nombre de tours dans `affiche_sous_chaine()` dépend des paramètres. Pour les suffixes, la première fois on fait 1 tour (couple (1,1)), puis la seconde 2 (couple (1,2)), etc. Pour les suffixes, on commence par le plus long (2,l) : $l-2$ tours, puis (3,l) : $l-1$ tours etc.

$$\begin{array}{ll} \text{préfixes} & 1 + 2 + 3 + \ldots + l-2 \\ \text{suffixes} & l-2 + l-3 + l-4 + \ldots + 1 \\ \hline & l-1 + l-1 + l-1 + \ldots + l-1 \quad = (l-2) \times (l-1) \end{array}$$

La complexité en nombre de tours est donc de $(l-2)(l-1) \approx O(l^2)$. L'algorithme proposé requiert par ailleurs $l+1$ cases mémoires pour le mot d'entrée.