

```

1: #include <stdlib.h>
2: #include "ab.h"
3:
4: //inicializa a árvore, fazendo a raiz igual a NULL
5: void cria(Arvore *A) {
6:     A->raiz=NULL;
7: }
8:
9: //verifica se a árvore está vazia
10: int IsEmpty(Arvore *A) {
11:     if (A->raiz==NULL)
12:         return 1;
13:     else return 0;
14: }
15:
16: //busca x na árvore, devendo iniciar com p=raiz ou outro ponto da árvore
17: no* busca(no *p, elem *x) {
18:     no *aux;
19:
20:     if (p==NULL)
21:         return(NULL);
22:     else if (p->info==*x)
23:         return(p);
24:     else {
25:         aux=busca(p->esq,x);
26:         if (aux==NULL)
27:             aux=busca(p->dir,x);
28:         return(aux);
29:     }
30: }
31:
32: //busca pai de x na árvore, devendo iniciar com p=raiz ou outro ponto da árvore
33: no* busca_pai(no *p, elem *x) {
34:     no *aux;
35:
36:     if (p==NULL)
37:         return(NULL);
38:     else if ((p->esq!=NULL) && (p->esq->info==*x))
39:         return(p);
40:     else if ((p->dir!=NULL) && (p->dir->info==*x))
41:         return(p);
42:     else {
43:         aux=busca_pai(p->esq,x);
44:         if (aux==NULL)
45:             aux=busca_pai(p->dir,x);
46:         return(aux);
47:     }
48: }
49:
50: //se possível, insere x à esquerda de pai na árvore
51: //se pai=-1, insere na raiz
52: void inserir_esq(Arvore *A, elem *x, elem *pai, int *erro) {
53:     no *aux, *p;
54:
55:     if (*pai!=-1) {
56:         aux=busca(A->raiz,pai);
57:         if ((aux!=NULL) && (aux->esq==NULL)) {
58:             p=(no*) malloc(sizeof(no));
59:             p->info=*x;
60:             p->esq=NULL;
61:             p->dir=NULL;
62:             aux->esq=p;

```

```

63:         *erro=0;
64:     }
65:     else *erro=1;
66: }
67: else if (A->raiz==NULL) {
68:     p=(no*) malloc(sizeof(no));
69:     p->info=*x;
70:     p->esq=NULL;
71:     p->dir=NULL;
72:     A->raiz=p;
73:     *erro=0;
74: }
75: else *erro=1;
76: }
77:
78: //se possível, insere x à direita de pai na árvore
79: //se pai=-1, insere na raiz
80: void inserir_dir(Arvore *A, elem *x, elem *pai, int *erro) {
81:     no *aux, *p;
82:
83:     if (*pai!=-1) {
84:         aux=busca(A->raiz,pai);
85:         if ((aux!=NULL) && (aux->dir==NULL)) {
86:             p=(no*) malloc(sizeof(no));
87:             p->info=*x;
88:             p->esq=NULL;
89:             p->dir=NULL;
90:             aux->dir=p;
91:             *erro=0;
92:         }
93:         else *erro=1;
94:     }
95:     else if (A->raiz==NULL) {
96:         p=(no*) malloc(sizeof(no));
97:         p->info=*x;
98:         p->esq=NULL;
99:         p->dir=NULL;
100:         A->raiz=p;
101:         *erro=0;
102:     }
103:     else *erro=1;
104: }
105:
106: //imprime os elementos da árvore
107: void imprimir(no *p) {
108:     if (p!=NULL) {
109:         printf("%d(",p->info);
110:         imprimir(p->esq);
111:         printf(",");
112:         imprimir(p->dir);
113:         printf(")");
114:     }
115:     else printf("null");
116: }
117:
118: //função para computar a altura de uma árvore binária; devo começar com p aponta
119: int altura(no *p) {
120:     int alt_esq, alt_dir;
121:
122:     if (p==NULL)
123:         return(0);
124:     else {

```

```

125:         alt_esq=1+altura(p->esq);
126:         alt_dir=1+altura(p->dir);
127:         if (alt_esq>alt_dir)
128:             return(alt_esq);
129:         else return(alt_dir);
130:     }
131: }
132:
133: //libera toda a memória usada na árvore
134: void finaliza(no *p) {
135:     if (p!=NULL) {
136:         finaliza(p->esq);
137:         finaliza(p->dir);
138:         free(p);
139:     }
140: }
141:
142: //função que remove um elemento da árvore, caso ele esteja nela
143: void remover(Arvore *A, elem *x, int *erro) {
144:     no *p, *pai;
145:     int eh_filho_esq;
146:
147:     //localizando o nó e o pai dele
148:     if ((A->raiz!=NULL) && (A->raiz->info==*x)) {
149:         pai=NULL;
150:         p=A->raiz;
151:     }
152:     else {
153:         pai=busca_pai(A->raiz,x);
154:         if (pai!=NULL) {
155:             if ((pai->esq!=NULL) && (pai->esq->info==*x)) {
156:                 p=pai->esq;
157:                 eh_filho_esq=1;
158:             }
159:             else {
160:                 p=pai->dir;
161:                 eh_filho_esq=0;
162:             }
163:         }
164:         else p=NULL;
165:     }
166:
167:     if (p==NULL)
168:         *erro=1;
169:     else {
170:         //primeiro caso: o nó não tem filhos
171:         if ((p->esq==NULL) && (p->dir==NULL)) {
172:             if (pai!=NULL) {
173:                 if (eh_filho_esq)
174:                     pai->esq=NULL;
175:                 else pai->dir=NULL;
176:             }
177:             else A->raiz=NULL;
178:             free(p);
179:             *erro=0;
180:         }
181:         //segundo caso: o nó tem um ou dois filhos
182:         else {
183:             if (p->esq!=NULL) {
184:                 p->info=p->esq->info;
185:                 p->esq->info=*x;
186:             }

```

```

187:         else {
188:             p->info=p->dir->info;
189:             p->dir->info=*x;
190:         }
191:         remover(A,x,erro);
192:     }
193: }
194: }
195:
196: //função de percurso de pre-ordem na árvore = busca em profundidade
197: void percurso_pre_ordem(no *p) {
198:     if (p!=NULL) {
199:         printf("%d\n",p->info);
200:         percurso_pre_ordem(p->esq);
201:         percurso_pre_ordem(p->dir);
202:     }
203: }
204:
205: //função de percurso de em-ordem na árvore
206: void percurso_em_ordem(no *p) {
207:     if (p!=NULL) {
208:         percurso_em_ordem(p->esq);
209:         printf("%d\n",p->info);
210:         percurso_em_ordem(p->dir);
211:     }
212: }
213:
214: //função de percurso de pos-ordem na árvore
215: void percurso_pos_ordem(no *p) {
216:     if (p!=NULL) {
217:         percurso_pos_ordem(p->esq);
218:         percurso_pos_ordem(p->dir);
219:         printf("%d\n",p->info);
220:     }
221: }
222:

```