Cursos de Verão 2010 - IME/USP
Tópicos de Programação
Àrvores binárias de busca
Prof. Alexandre da Silva Freire

1 Árvores binárias de busca

Utilizaremos a seguinte estrutura para representar os nós de uma árvore binária:

```
typedef struct No {
   int elem;
   struct No* esq; /* ponteiro para o filho esquerdo */
   struct No* dir; /* ponteiro para o filho direito */
   struct No* pai; /* ponteiro para o pai */
} No;

No* novoNo(int elem) {
   No* no = malloc(sizeof(No));
   no->elem = elem;
   no->esq = NULL;
   no->dir = NULL;
   no->pai = NULL;
   return no;
}
```

Utilizaremos a seguinte estrutura para representar uma árvore binária:

```
typedef struct ArvBin {
   No* raiz; /* ponteiro para a raiz da árvore */
} ArvBin;

ArvBin* novaArv() {
   ArvBin* arv = malloc(sizeof(ArvBin));
   arv->raiz = NULL;
   return arv;
}
```

Uma *árvore binária de busca* é uma árvore binária tal que cada nó x tem a seguinte propriedade: o campo *elem* é

- maior ou igual ao campo *elem* de qualquer nó na subárvore esquerda de x e
- menor do que o campo *elem* de qualquer nó na subárvore direita de x.

Em outras palavras, `y->elem` $\leq$ `x->elem` $<$ `z->elem`, para todo nó `y` na subárvore esquerda de `x` e todo nó `z` na subárvore direita de `x`.

Podemos percorrer uma árvore binária de várias formas diferentes. Vejamos três tipos de percurso diferentes:

- Percurso *em ordem*:

```
void em_ordem(ArvBin* arv) {
   em_ordem_rec(arv->raiz);
}

void em_ordem_rec(No* no) {
   if(no != NULL) {
      em_ordem_rec(no->esq);
      printf("%d\n", no->elem);
      em_ordem_rec(no->dir);
   }
}
```

- Percurso *pré-ordem*:

```
void pre_ordem(ArvBin* arv) {
   pre_ordem_rec(arv->raiz);
}

void pre_ordem_rec(No* no) {
   if(no != NULL) {
      printf("%d\n", no->elem);
      pre_ordem_rec(no->esq);
      pre_ordem_rec(no->dir);
   }
}
```

- Percurso *pós-ordem*:

```
void pos_ordem(ArvBin* arv) {
   pos_ordem_rec(arv->raiz);
}

void pos_ordem_rec(No* no) {
   if(no != NULL) {
      pos_ordem_rec(no->esq);
      pos_ordem_rec(no->dir);
      printf("%d\n", no->elem);
   }
}
```

Para obter um elemento mínimo/máximo em uma árvore binária, podemos utilizar as seguintes funções:

```
No* minimo(ArvBin* arv) {
   return minimo_rec(arv->raiz);
}

No* minimo_rec(No* no) {
   if(no->esq == NULL) {
      return no;
   } else {
      return minimo_rec(no->esq);
   }
}

No* maximo(ArvBin* arv) {
   return maximo_rec(arv->raiz);
}

No* maximo_rec(No* no) {
   if(no->dir == NULL) {
      return no;
   } else {
      return maximo_rec(no->dir);
   }
}
```

A *altura* de um nó x em uma árvore binária é a distância entre x e o seu descendente mais afastado. Mas precisamente, a altura de x é o número de passos do mais longo caminho que leva de x até uma folha (a propósito, uma *folha* é um nó que não tem filhos). A altura de uma árvore é a altura de sua raiz. Uma árvore com um único nó tem altura 0. Para obter a altura de uma árvore binária, podemos utilizar a seguinte função (não esqueça da função de *embrulho*, a qual apenas devolve o resultado da chamada `altura_rec(arv->raiz);`):

```
int altura_rec (No* no) {
   if (no == NULL) {
      return -1;
   } else {
      int altEsq = altura_rec(no->esq);
      int altDir = altura_rec(no->dir);

      if (altEsq < altDir) {
         return altDir + 1;
      } else {
         return altEsq + 1;
      }
   }
}
```

Para obter o sucessor/predecessor (considerando o percurso *em ordem*) de um nó em uma árvore binária, podemos utilizar as seguintes funções:

```
/* se "no" não tiver sucessor, devolve NULL */
int sucessor (No* no) {
   if (no->dir != NULL) {
      return minimo_rec(no->dir);
   }
   while (no->pai != NULL && no->pai->dir == no) {
      no = no->pai;
   }
   return x->pai;
}

/* se "no" não tiver predecessor, devolve NULL */
int predecessor (No* no) {
   if (no->esq != NULL) {
      return maximo_rec(no->esq);
   }
   while (no->pai != NULL && no->pai->esq == no) {
      no = no->pai;
   }
   return x->pai;
}
```

Para obter o endereço do nó no qual encontra-se um elemento e, podemos utilizar a seguinte função de *busca*:

```
/* se o elemento "e" não estiver na árvore, devolve NULL */
No* busca(No* no, int e) {
    if(no == NULL || no->elem == e) {
       return no;
    } else if(e < no->elem) {
       return busca(no->esq, e);
    } else {
       return busca(no->dir, e);
    }
}
```

Para inserir na árvore um elemento e, podemos utilizar a seguinte função:

```
void insere(ArvBin* arv, int e) {
   No* p;
   No* f;
   No* novo = novoNo(e);
   if(arv->raiz == NULL) {
      arv->raiz = novo;
   } else {
      f = arv->raiz;
      while(f != NULL) {
         p = f;
         if(e <= f->elem) {
            f = f->esq;
         } else {
            f = f->dir;
         }
      }
      novo->pai = p;
      if(e <= p->elem) {
         p->esq = novo;
      } else {
         p->dir = novo;
      }
   }
}
```

Para remover um nó qualquer da árvore, podemos utilizar a seguinte função:

```
/*
 *  assumimos que o "no" não é NULL e que
 *  o "no" está na árvore "arv".
 */
void removeNo(ArvBin* arv, No* no) {
   No* x;

   if(no->esq == NULL && no->dir == NULL) {
       if(no->pai == NULL) {
          arv->raiz = NULL;
       } else {
          if(no->pai->esq == no) {
             no->pai->esq = NULL;
          } else {
             no->pai->dir = NULL;
          }
       }
       free(no);
   } else {
      if(no->dir != NULL) {
         x = sucessor(no);
      } else {
         x = predecessor(no);
      }

      no->elem = x->elem;
      removeNo(arv, x);
   }
}
```

Seja A uma árvore binária de busca com n nós, e seja h a altura de A. Vejamos o consumo de tempo dos algoritmos apresentados até aqui, considerando a árvore A:

Algoritmo	**Consumo de tempo**
`em_ordem`	$\Theta(n)$
`pre_ordem`	$\Theta(n)$
`pos_ordem`	$\Theta(n)$
`altura`	$\Theta(n)$
`insere`	$O(h)$
`removeNo`	$O(h)$
`minimo`	$O(h)$
`maximo`	$O(h)$
`sucessor`	$O(h)$
`predecessor`	$O(h)$
`busca`	$O(h)$