Лабораторная работа № 9

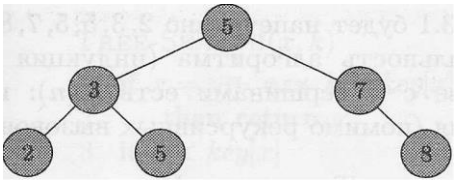
Двоичные деревья поиска

Теоретическая часть

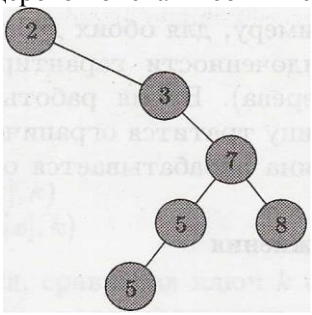
Двоичное дерево (binary tree) можно определить рекурсивно как конечный набор вершин, который
a) либо пуст (не содержит вершин),
b) либо разбит на три непересекающиеся части: вершину, называемую **корнем** (root), двоичное дерево, называемое **левым поддеревом** (left subtree) корня, и двоичное дерево, называемое **правым поддеревом** (right subtree) корня.

Двоичное дерево, не содержащее вершин, называется **пустым** (empty). Если левое поддерево непусто, то его корень называется **левым ребёнком** (left child) корня всего дерева; **правый ребёнок** (right child) определяется аналогично. Если левое или правое поддерево корня пусто, то говорят, что у корня нет левого или правого ребёнка (child is absent).

В **двоичном дереве поиска** (binary search tree - BST; пример приведён на рисунке 1) каждая вершина может иметь (или не иметь) левого и правого ребёнка; каждая вершина, кроме корня, имеет родителя. При представлении с использованием указателей надо хранить для каждой вершины дерева, помимо значения ключа *key* и дополнительных данных, также и указатели *left, right* и *p* (левый ребёнок, правый ребёнок, родитель). Если ребёнка (или родителя – для корня) нет, соответствующее поле содержит NULL.



(а) Двоичное дерево поиска высоты 2 с 6 вершинами.



(б) Менее эффективное дерево высоты 4, содержащее те же ключи.

Рисунок 1 - Двоичные деревья поиска.

Левое поддерево произвольной вершины *x* содержит ключи, не превосходящие *key*[*x*], правое – не меньшие *key*[*x*]. Разные двоичные деревья поиска могут представлять одно и то же множество. Время выполнения (в худшем случае) большинства операций пропорционально высоте дерева.

Двоичные деревья поиска позволяют выполнять следующие операции с динамическими множествами: SEARCH (поиск), MINIMUM (минимум), MAXIMUM (максимум), PREDECESSOR (предыдущий), SUCCESSOR (следующий), INSERT (вставить) и DELETE (удалить). Таким образом, дерево поиска может быть использовано как словарь.

Ключи в двоичном дереве поиска хранятся с соблюдением **свойства упорядоченности** (binary-search-tree property). Пусть x – произвольная вершина двоичного дерева поиска. Если вершина y находится в левом поддереве вершины x, то $key[y] <= key[x]$. Если y находится в правом поддереве x, то $key[y] >= key[x]$.

Так, на рисунке 1 (а) в корне дерева хранится ключ 5, ключи 2, 3 и 5 в левом поддереве корня не превосходят 5, а ключи 7 и 8 в правом – не меньше 5. То же самое верно для всех вершин дерева. Например, ключ 3 на рисунке 1 (а) не меньше ключа 2 в левом поддереве и не больше ключа 5 в правом.

Свойство упорядоченности позволяет напечатать все ключи в неубывающем порядке с помощью простого рекурсивного алгоритма (называемого по-английски inorder tree walk). Этот алгоритм печатает ключ корня поддерева после всех ключей его левого поддерева, но перед ключами правого поддерева. (Заметим, что порядок, при котором корень предшествует обоим поддеревьям, называется preorder; порядок, в котором корень следует за ними, называется postorder.) Центрированный обход дерева T реализуется процедурой INORDER-TREE-WALK($root[T]$):

```
INORDER-TREE-WALK(x)
   1. if x <> NULL
   2. then INORDER-TREE-WALK(left[x])
   3.      напечатать key[x]
   4.      INORDER-TREE-WALK(right[x]
```

К примеру, для обоих деревьев на рисунке 1 будет напечатано 2, 3, 5, 5, 7, 8.

Поиск в двоичном дереве

Процедура поиска получает на вход искомый ключ k и указатель x на корень поддерева, в котором производится поиск. Она возвращает указатель на вершину с ключом k (если такая есть) или NULL (если такой вершины нет).

```
TREE-SEARCH(x, k)
   1. if x = NULL или k = key[x]
   2. then return x
   3. if k < key[x]
   4. then return TREE-SEARCH(left[x], k)
   5. else return TREE-SEARCH(right[x], k)
```

В процессе поиска мы двигаемся от корня, сравнивая ключ k с ключом, хранящимся в текущей вершине x. Если они равны, поиск завершается. Если $k < key[x]$, то поиск продолжается в левом поддереве x (ключ k может быть только там, согласно свойству упорядоченности). Если $k > key[x]$, то поиск продолжается в правом поддереве. Длина пути поиска не превосходит высоты дерева. Так, на рисунке 2 для поиска ключа 13 мы идём от корня по пути 15 -> 6 -> 7 -> 13.

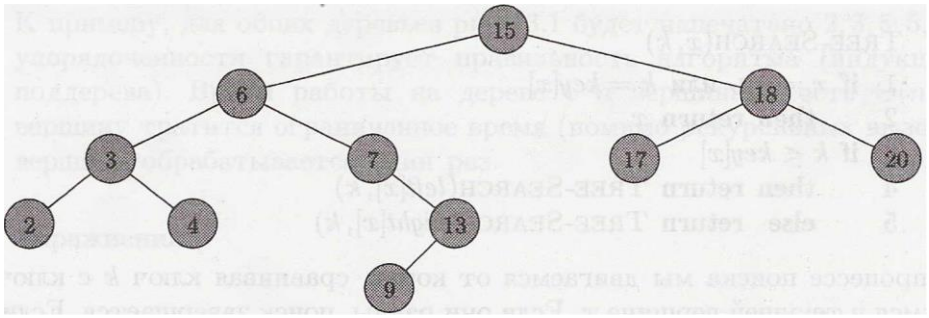


Рисунок 2 - Поиск в двоичном дереве

Вот итеративная версия той же процедуры (которая, как правило, более эффективна):

```
ITERATIVE-TREE-SEARCH(x, k)
```

```
   1. while x <> NULL и k <> key[x]
   2. do if k < key[x]
   3.    then x <- left[x]
   4.    else x <- right[x]
   5. return x
```

Минимум и максимум

Минимальный ключ в дереве поиска можно найти, пройдя по указателям *left* от корня до тех пор, пока не встретится NULL. Так, на рисунке 2 чтобы найти минимальный ключ 2, мы всё время идём налево; чтобы найти максимальный ключ 20 - направо. Для вершины с ключом 15 следующей будет вершина с ключом 17 (это минимальный ключ в правом поддереве вершины с ключом 15). У вершины с ключом 13 нет правого поддерева; поэтому, чтобы найти следующую за ней вершину, мы поднимаемся вверх, пока не пройдём по ребру, ведущему вправо-вверх; в данном случае у следующей вершины - ключ 15. Реализация описанного алгоритма выглядит так:

```
TREE-MINIMUM(x)
   1. while left[x] <> NULL
   2. do x <- left[x]
   3. return x
```

Свойство упорядоченности гарантирует правильность процедуры TREE-MINIMUM. Если у вершины *x* нет левого ребёнка, то минимальный элемент поддерева с корнем *x* есть *x*, так как любой ключ в правом поддереве не меньше *key*[*x*]. Если же левое поддерево вершины *x* не пусто, то минимальный элемент поддерева с корнем *x* находится в этом левом поддереве (поскольку сам *x* и все элементы правого поддерева больше). Алгоритм поиска максимального элемента дерева симметричен алгоритму поиска минимального элемента:

```
TREE-MAXIMUM(x)
   1. while right[x] <> NULL
   2. do x <- right[x]
   3. return x
```

Следующий и предыдущий элементы

Как найти в двоичном дереве элемент, следующий за данным элементом? Свойство упорядоченности позволяет сделать это, двигаясь по дереву. Вот процедура, которая возвращает указатель на следующий за *x* элемент (если все ключи различны, он содержит следующий по величине ключ) или NULL, если элемент *x* последний в дереве:

```
TREE-SUCCESSOR( x)
   1. if right[x] <> NULL
   2. then return TREE-MINIMUM(right[x])
   3. y <- p[x]
   4. while y <> NIL и x = right[y]
   5. do x <- y
   6.    y <- p[y]
   7. return y
```

Процедура TREE-SUCCESSOR отдельно рассматривает два случая.

a) Если правое поддерево вершины *x* непусто, то следующий за *x* элемент – является крайним левым узлом в правом поддереве, который обнаруживается в строке 2 вызовом процедуры TREE-MINIMUM(*right*[*x*]). Например, на рисунке 2 за вершиной с ключом 15 следует вершина с ключом 17.
b) Пусть теперь правое поддерево вершины *x* пусто. Тогда мы идём от *x* вверх, пока не найдём вершину, являющуюся левым сыном своего родителя (строки 3-7). На рисунке 2 следующим за узлом с ключом 13 является узел с ключом 15.

Процедура TREE-PREDECESSOR симметрична.

Добавление элемента

Процедура TREE-INSERT добавляет заданный элемент в подходящее место дерева *T* (сохраняя свойство упорядоченности). Параметром процедуры является указатель *z* на новую вершину, в которую помещены значения *key*[*z*] (добавляемое значение ключа), *left*[*z*] = NIL и *right*[*z*] = NULL. В ходе работы процедура меняет дерево *T* и (возможно) некоторые поля вершины *z*, после чего новая вершина с данным значением ключа оказывается вставленной в подходящее место дерева.

```
TREE-INSERT(T, z)
   1. y <- NULL
   2. x <- root[T]
   3. while x <> NULL
   4. do y <- x
   5.    if key[z] < key[x]
   6.    then x <- left[x]
   7.    else x <- right[x]
   8. p[z] <- y
   9. if y = NULL
   10.     then root[T] <- z
   11.     else if key[ z] < key[y]
   12.          then left[y] <- z
   13.          else right[y] <- z
```

На рисунке 3 показано, как работает процедура TREE-INSERT. Подобно процедурам TREE-SEARCH и ITERATIVE-TREE-SEARCH, она двигается вниз по дереву, начав с его корня. При этом в вершине *y* сохраняется указатель на родителя вершины *x* (цикл в строках 3-7). Сравнивая *key*[*z*] с *key*[*x*], процедура решает, куда идти - налево или направо. Процесс завершается, когда *x* становится равным NULL. Этот NULL стоит как раз там, куда надо поместить *z*, что и делается в строках 8-13.

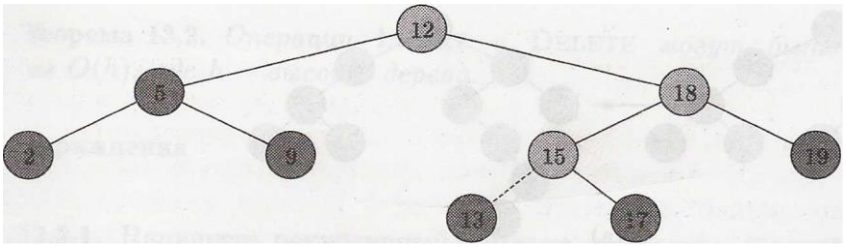


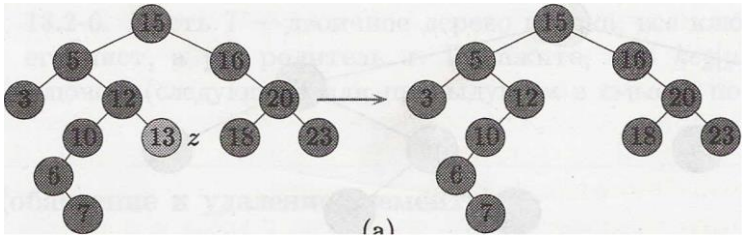
Рисунок 3 - Добавление элемента с ключом 13

Светло-серые вершины находятся на пути от корня до позиции нового элемента. Пунктир связывает новый элемент со старыми элементами.

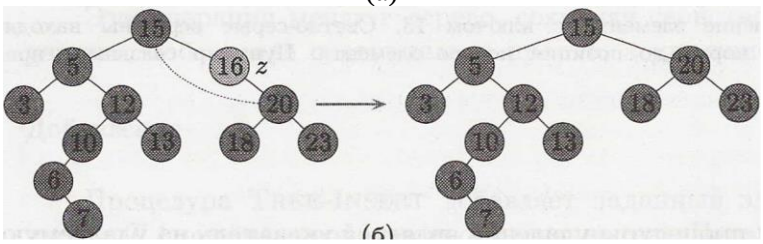
Удаление элемента

Параметром процедуры удаления является указатель на удаляемую вершину. При удалении возможны три случая, показанные на рисунке 4.

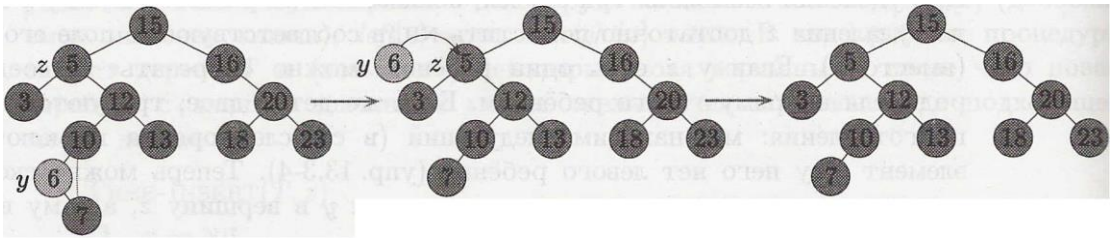
a) Если у *z* нет детей, для удаления *z* достаточно поместить NULL в соответствующее поле его родителя (вместо *z*).
b) Если у *z* есть один ребёнок, можно «вырезать» *z*, соединив его родителя напрямую с его ребёнком.
c) Если же детей двое, требуются некоторые приготовления: мы находим следующий (в смысле порядка на ключах) за *z* элемент *y*; у него нет левого ребёнка. Теперь можно скопировать ключ и дополнительные данные из вершины *y* в вершину *z*, а саму вершину *y* удалить описанным выше способом.



(а)



(б)



(в)

Рисунок 4 - Удаление вершины *z* из двоичного дерева поиска

Примерно так и действует процедура TREE-DELETE (хотя рассматривает эти три случая в несколько другом порядке).

```
TREE-DELETE(T, z)
   1. if left[z] = NULL или right[z] = NULL
   2. then y <- z
   3. else y <- TREE-SUCCESSOR(Z)
   4. if left[y] <> NULL
   5. then x <- left[y]
   6. else x <- right[y]
   7. if x <> NULL
   8. then p[x] <- p[y]
   9. if p[y] = NULL
   10.      then root[T] <- x
   11.      else if y = left[p[y]]
   12.           then left[p[y]] <- x
   13.           else right[p[y]] <- x
   14.      if y <> z
   15.      then key[z] <- key[y]
   16.      // Копируем дополнительные данные, связанные с y.
   17.      return y
```

В строках 1-3 определяется вершина *y*, которую мы потом вырежем из дерева. Это либо сама вершина *z* (если у *z* не более одного ребёнка), либо следующий за *z* элемент (если у *z* двое детей). Затем в строках 4-6 переменная *x* становится указателем на существующего ребёнка вершины *y*, или равной NULL, если у *y* нет детей. Вершина *y* вырезается из дерева в строках 7-13 (меняются указатели в вершинах *p*[*y*] и *x*). При этом отдельно рассматриваются граничные случаи, когда *x* = NULL или когда *y* является корнем дерева. Наконец, в строках 14-16, если вырезанная вершина *y* отлична от *z*, ключ (и дополнительные данные) вершины *y* перемещаются в *z* (ведь

нам надо было удалить *z*, а не *y*). Наконец, процедура возвращает указатель *y* (это позволит вызывающей процедуре впоследствии освободить память, занятую вершиной *y*).

Практическая часть

Простой пример построения и печати двоичного дерева поиска приведен в файле REF2PTR.CPP. В программе можно отметить три особенности:

1) Использование параметров командной строки.
2) Использование ссылок на указатели (в C99 этого может не быть).
3) Построение двоичного дерева поиска и его обход в обратном порядке.

Задание на лабораторную работу

Написать программу, которая читает данные из лабораторной работы «Алгоритмы внутренней сортировки» и строит двоичное дерево поиска с выбранными вами ключами. Для построенного дерева реализовать три варианта обхода дерева и следующие операции: SEARCH (поиск), MINIMUM (минимум), MAXIMUM (максимум), PREDECESSOR (предыдущий), SUCCESSOR (следующий), INSERT (вставить) и DELETE (удалить).

Рекомендуемая литература

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 2-е изд. – М.: Вильямс, 2009. 960 с. (Глава 12)