ЛАБОРАТОРНАЯ РАБОТА №10

По дисциплине: Алгоритмы и структуры данных
Тема работы: Бинарное дерево поиска (BST)
Цель работы: реализовать программно хранение данных посредством бинарного дерева поиска, описать операции, модифицирующие множество, и запросы.
Количество часов: 4

Содержание работы:

1. Описание структуры данных
2. Описание главной функции
3. Добавление функций основных операций
4. Анализ времени работы основных операций для разных наборов входных чисел
5. Выводы

Методические указания по выполнению

Описание структуры данных

Бинарное дерево поиска

Бинарное или двоичное дерево – это конечное множество элементов, которое либо пусто, либо содержит один элемент, называемый корнем дерева, а остальные элементы множества делятся на два непересекающихся подмножества, каждое из которых само является двоичным деревом. Эти подмножества называют соответственно левым и правым поддеревьями.

Каждая вершина дерева – запись с несколькими полями: ключ (*x.key*), указатель на родителя (*x.p*), указатель на левого сына (*x.left*) и указатель на правого сына (*x.right*).

Бинарное дерево поиска подчиняется следующему **свойству** упорядоченности:

пусть x произвольная вершина дерева, если y – левый ребенок, то x.key >= y.key, если y вершина правого поддерева, то x.key <= y.key.

Время работы всех операций пропорционально высоте дерева и равно *O(lg n).*

Поиск элемента с заданным ключом в бинарном дереве поиска

Рассмотрим задачу поиска элемента с заданным ключом в бинарном дереве поиска. Функция поиска в бинарном дереве *Tree_Search()* получает на вход искомый ключ и указатель на вершину поддерева, в котором нужно искать. В процессе поиска мы двигаемся от корня, сравнивая ключ *k* с ключом, хранящимся в текущей вершине *x*. Если они равны, поиск завершается. Если *k* < *x.key*, то поиск продолжается в левом поддереве *x* (ключ *k* может быть только там, согласно свойству упорядоченности). Если *k* > *x.key*, то поиск продолжается в правом поддереве. Функция возвращает указатель на вершину с ключом *k* или *nil*, если такой вершины нет.

```
Tree_Search(x, k)
1. if x == nil or k == x.key
2.    return x
3. if k < x.key
4.    return Tree_Search(x.left, k)
5.    else return Tree_Search(x.right, k)
```

Минимальный ключ в дереве поиска

Минимальный ключ в дереве поиска можно найти, пройдя по указателям *left* от корня (пока не спустимся до *nil*). Функция *Tree_Minimum()* возвращает указатель на минимальный элемент поддерева с корнем *x*.

```
Tree_Minimum(x)
1. while x.left ≠ nil
2.     x = x.left
3. return x
```

Элемент, следующий за данным

Свойство упорядоченности позволяет найти элемент, следующий за данным (в смысле значения ключа). Ниже приведена функция, которая возвращает указатель на следующий за *x* элемент (если все ключи различны, он содержит следующий по величине ключ) или *nil*, если элемент *x* – последний в дереве. Функция *Tree_Successor* отдельно рассматривает два случая. Если правое поддерево вершины x непусто, то следующий за *x* элемент – минимальный элемент в этом поддереве и равен *Tree_Minimum(x.right)*.

Пусть теперь правое поддерево вершины *x* пусто. Тогда мы идём от *x* вверх, пока не найдём вершину, являющуюся левым сыном своего родителя. Этот родитель (если он есть) и будет искомым элементом.

```
Tree_Successor(x)
1. if x.right ≠ nil
2.     return Tree_Minimum(x.right)
3. y = x.p
4. while y ≠ nil and x == y.right
5.     x = y
6.     y = y.p
7. return y
```

Добавление элемента в бинарное дерево

Добавление элемента в бинарное дерево происходит с сохранением свойства упорядоченности. Ниже приводится функция *Tree_Insert()*, которая добавляет заданный элемент *z* (указатель на новый элемент) в дерево *T*. В ходе работы функция меняет дерево и некоторые поля вершины *z*, после чего новая вершина с данным значением ключа оказывается вставленной в подходящее место.

```
Tree_Insert(T,z)
1. y = nil
2. x = T
3. while x ≠ nil
4.    y=x
5.    if z.key < x.key
6.          x = x.left
7.    else x = x.right
8. z.p = y
9. if y == nil
10.    T = z
11.    else if z.key < y.key
12.              y.left = z
13.         else y.right = z
```

Изменить код функции так, чтобы инициализация узла *z* производилась в начале функции.

Удаление вершины дерева

Удаление вершины дерева производится с помощью функции *Tree_Delete()*. Параметром функции удаления является указатель на удаляемую вершину. При удалении возможны три случая:

1) если *z* не имеет левого дочернего узла, то *z* заменяется правым дочерним узлом. Если правый дочерний также *nil*, то у *z* детей нет (у родительского узла указатель на дочерний узел станет *nil*);
2) если *z* не имеет правого дочернего узла, то *z* заменяется левым дочерним узлом;
3) если *z* имеет оба дочерних узла, необходимо найти узел *y*, следующий за *z*. Он расположен в правом поддереве *z*, и не имеет левого дочернего узла. Нужно его вырезать из текущего местоположения, и вставить на место *z*. Здесь возможны варианты:
   - если *y* является правым дочерним узлом *z*, то он становится на место *z*. При этом левый ребенок *z* становится левым ребенком *y*;
   - иначе узел *y* расположен в правом поддереве, но не является правым дочерним узлом *z*. В этом случае сначала узел *y* заменяется своим правым дочерним узлом. И только затем становится на место *z*, принимая детей *z* в качестве своих.

```
Tree_Delete(T,z)
1. if z.left == nil
2.     Transplant(T,z,z.right)
3. else if z.right == nil
4.         Transplant(T,z,z.left)
5.     else y = Tree_Minimum(z.right)
6.          if y.p ≠ z
7.               Transplant(T,y,y.right)
8.               y.right = z.right
9.               y.right.p = y
10.         Transplant(T,z,y)
11.         y.left = z.left
12.         y.left.p = y
```

```
Transplant(T,u,v)
1. if u.p == nil
2.           T = v
3. else if u == u.p.left
4.          u.p.left = v
5.           else u.p.right = v
6. if v ≠ nil
7.           v.p = u.p
```

Для перемещения поддеревьев, сначала определите функцию *Transplant()*. Функция заменяет одно дочернее поддерево с корнем в узле *u* другим поддеревом с корнем в узле *v*. При этом родитель узла *u* становится родителем узла *v*, который в свою очередь становится дочерним по отношению к *u*.

Здесь возможны такие случаи:

- если узел *u* является корнем дерева, то узел *v* станет новым корнем (строки 1-2);
- иначе узел *u* является левым (строки 3-4) или правым ребенком (строка 5), и обновляется соответствующий родительский указатель.
- если узел *v* не является пустым, то его указатель на родителя также обновляется (строки 6-7).

Задачи

1. Реализуйте функцию центрированного обхода узлов бинарного дерева поиска.
2. Опишите функцию поиска предшественника узла *x Tree_Predecessor(x)*.
3. Постройте бинарные деревья поиска высотой 3, 4, 6 для множества ключей {1,2,4,5,7,8,9,12}.

Пособия и инструменты

1. MS Visual Studio 2010/…/1017
2. Data Structure Visualizations. [Электронный ресурс] – Режим доступа: http://www.cs.usfca.edu/~galles/visualization/java/download.html.

Вопросы для защиты лабораторной работы

1. Дайте асимптотическую оценку выполнения каждой операции.
2. Каким будет асимптотическое время работы функции *Tree_Insert()* при вставке *m* одинаковых элементов в бинарное дерево поиска.
3. Какова высота бинарного дерева поиска, построенного из *n* узлов случайным образом.

Литература

1. Кормен Т.Х. Алгоритмы: построение и анализ, 3-е издание : Пер. с англ. / Т.Х. Кормен, Ч.И. Лейзерсон, Р.Л. Ривест, К. Штайн. – М.: Издательский дом «Вильямс», 2013. – 1328 с.
2. Algorithms and Data Structures with implementations in Java and C++ [Electronic Resource]. – URL: http://www.algolist.net/.
3. Data Structure Visualizations / David Galles, Department of Computer Science // University of San Francisco [Electronic Resource]. – URL: http://www.cs.usfca.edu/~galles/visualization/Algorithms.html.
4. Анимированные визуализации структур данных / VISUALGO [Electronic Resource]. – URL: http://ru.visualgo.net/.