Описание используемых структур данных

«Англо-русский словарь с использованием бинарного дерева поиска» (1.1.1.)

Студент:
Потапова Алина

Группа:
3530202/90002

На данном этапе курсовой работы я подробно опишу используемые для словаря классы, а конкретно – бинарное дерево поиска (для работы со словами на английском языке) и односвязный список (для хранения русских слов-переводов или целых чисел). Для удобства работы с различными типами, все рассматриваемые мной классы будут шаблонными.
Для удобства воспользуюсь сокращениями:

1. ‘голова’ – первый узел односвязного списка;
2. ‘хвост’ – последний узел списка;
3. БДП – бинарное дерево поиска.

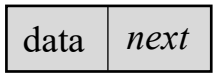
Класс Односвязный список

- **Идентификатор доступа private:**

1. Структура узла списка (Node);
2. Указатель на ‘голову’ списка (тип – указатель на узел – Node*);
3. Указатель на ‘хвост’ списка (тип – указатель на узел – Node*);
4. Метод вставки узла в начало списка (тип – void; параметр – указатель на узел);
5. Метод вставки узла в конец списка (тип – void; параметр – указатель на узел).

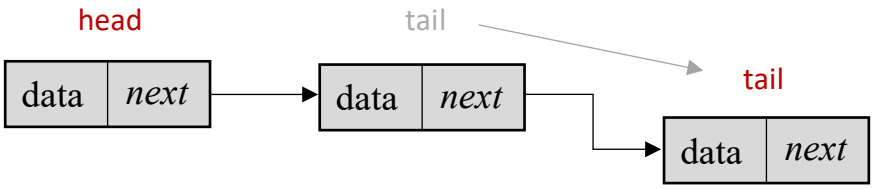
Описание структуры узла списка (Node)

Каждый узел односвязного списка имеет два поля *data* и *next*. Поле *data* – это слово-перевод (шаблонный тип - T). В поле *next* хранится адрес узла (тип – Node*), следующего за текущим.



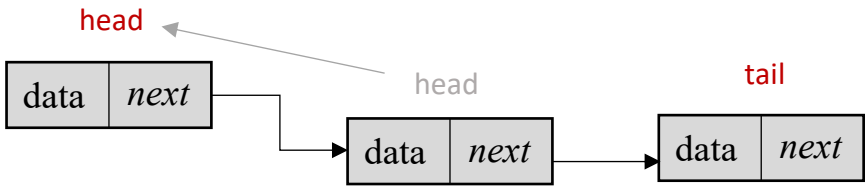
Метод вставки узла в конец списка

```
указатель на узел <- x

INSERT_TAIL(x)
{
    if список не пуст
        следующий за текущим хвостом <- x
        хвост списка <- x
    else
        хвост списка <- x
        голова списка <- x
}
```



Метод вставки узла в начало списка

```
указатель на узел <– x

INSERT_HEAD(x)
{
    приравнять указатель на следующий элемент x к голове списка
    if список пуст
        хвост списка <– x
    голова списка <– x
}
```



- **<u>Идентификатор доступа public:</u>**

1. Конструктор без параметров;
2. Конструктор копирования;
3. Деструктор;
4. Метод добавления слова в начало списка (тип – void; параметр – слово);
5. Метод добавления слова в конец списка (тип – void; параметр – слово);
6. Метод, определяющий количество слов в списке (тип – целочисленный);
7. Метод вывода слов-переводов (через запятую);
8. Метод ввода слов-переводов (для ввода из файла);
9. Метод, определяющий пустой список или нет (тип – bool).

Конструктор копирования

```
ссылка на копируемый объект <- list
LIST(list)
{
    инициализация указателя на головной узел list <- temp
    while temp не пуст
    {
        добавить слово в temp в конец списка
        переместить temp на следующий узел
    }
}
```

Деструктор

```
~LIST()
{
    while голова не пуста
    {
        инициализация указателя на следующий за головным узлом
            <- temp
        удалить голову
        новая голова списка <- temp
    }
}
```

Метод добавления слова в конец списка

```
слово для вставки <- data

PUSH_BACK(data)
{
    if список пуст
    {
        голова списка <- узел со словом для вставки
    }
    else
    {
        инициализация указателя на головной(текущий) узел ->
            current
        while current не хвост
        {
            переместить current на следующий узел
        }
        следующий за current <- узел со словом для вставки
    }
}
```

Метод добавления слова в начало списка

```
слово для вставки <- data
PUSH_FRONT(data)
{
    голова списка <- узел со словом для вставки
}
```

Метод, определяющий количество слов в списке

```
GET_SIZE()
{
    инициализация целочисленной переменной, отвечающей за размер
       списка <- size
    for голова списка...хвост списка
    {
       увеличить size на 1
    }
    вернуть size
}
```

Метод вывода слов-переводов (через запятую)

```
ссылка на стандартный поток вывода <- out

PRINT(out)
{
    инициализация указателя на узел <- node
    for голова списка...хвост списка (перемещать node на
       следующий узел)
    {
       if следующий от node пуст
            out <- слово в node
       else
            out <- слово в node <- запятая
    }
}
```

Метод, определяющий пустой список или нет

```
IS_EMPTY()
{
    if голова списка пустая (список пуст)
       вернуть истину
    else
       вернуть ложь
}
```

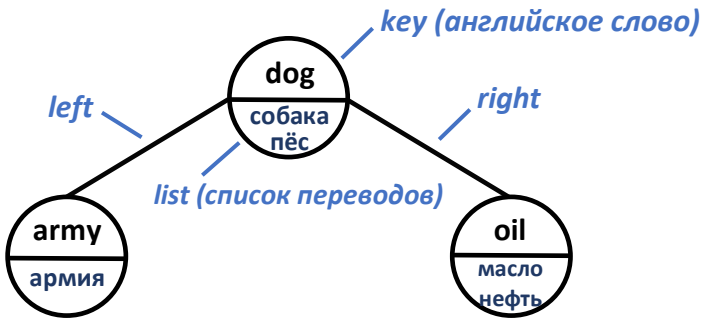
Класс Бинарное Дерево Поиска

- **Идентификатор доступа private:**

1. Структура узла дерева;
2. Указатель на вершину дерева (тип – Node*);
3. Поиск узла в дереве по слову (тип – Node*; параметр – слово);
4. Поиск следующего слова (тип – Node*; параметр – слово);
5. Поиск минимального следующего узла (тип – Node*; параметр – указатель на узел);
6. Определение количества слов (тип – целочисленный тип: параметр – указатель на узел);
7. Рекурсивное освобождение памяти (тип – void; параметр – указатель на узел);
8. Вывод информации, хранящейся в узле (тип – void; параметры – ссылка на стандартный поток ввода, указатель на узел).
9. Удаление узла из дерева, не нарушающее порядка элементов (тип – void; параметр – указатель на узел).

Описание структуры узла дерева (Node)

Каждый узел БДП имеет два поля *list, key, p, left и right*. Поле *key* – это слово на английском языке. Поле list – список слов-переводов (тип – односвязный список). В поле *left* хранится адрес узла (тип – Node*), следующего слева за текущим. В поле *right* хранится адрес узла (тип – Node*), следующего справа за текущим. В поле *p* хранится адрес родительского узла (тип – Node*). Для наглядности:



Определение количества узлов (рекурсивно)

```
указатель на узел <– node

GET_COUNT_SUB_TREE(node)
{
    if node не пуст
        вернуть 0;
    вернуть 1 + количество узлов в левом узле от node +
        количество узлов в правом узле от node
}
```

Поиск узла в дереве по слову

```
слово для поиска -> key

SEARCH_NODE(key)
{
     инициализация указателя на головной(текущий) узел -> current
     инициализация логической переменной, определяющей найдено ли
        слово в словаре, по умолчанию значение false -> find

    while текущий не пуст и слово не найдено
    {
        if key совпадает со словом в current (слово найдено)
        {
            присвоить переменной find значение true
        }
        else if key меньше слова в current
        {
            переместить current влево
        }
        else
        {
            переместить current вправо
        }
    }
    if истинное значение find (слово найдено)
    {
        вернуть current
    }
    else
    {
        вернуть nullptr
    }
}
```

Рекурсивное освобождение памяти

```
указатель на узел <- node

DELETE_SUB_TREE(Node* node)
{
    while node не пуст
    {
         освобождение памяти для левого от текущего узла
         освобождение памяти для правого от текущего узла
         удалить node
         приравнять node к nullptr
    }
}
```

Поиск следующего слова

```
слово для поиска следующего -> key

FIND_NEXT(key)
{
     инициализация указателя на узел с данными key -> current
     инициализация указателя на следующий узел -> nextNode
     if current пуст
     {
          вернуть пустой указатель
     }
     if правый от current не пуст
     {
          вернуть минимальный от правого current
     }
     nextNode <- родитель current
     while nextNode не пуст и current равен правому от nextNode
     {
          current <- nextNode
          nextNode <- родитель nextNode
     }
    вернуть nextNode
}
```

Поиск минимального следующего узла

```
указатель на узел <- node

FIND_MIN(node)
{
     if список пуст
     {
          вернуть пустой указатель
     }
     while левый от node не пуст
     {
          node <- левый от node
     }
     вернуть node
}
```

Вывод информации, хранящейся в узле (рекурсивно)

```
ссылка на стандартный поток вывода <– out
указатель на вершину дерева <– root

PRINT_NODE(out, root)
{
    out <– открытая скобка
    if в root есть данные
    {
       out <– данные из root
       вывод информации, хранящейся в левом от root узле
       вывод информации, хранящейся в правом от root узле
    }
    out <– закрытая скобка

}
```

Удаление узла из дерева, не нарушающее порядка элементов

```
указатель на узел <– node

DELETE_NODE(Node* node)
{
    if node пуст
    {
        вывод сообщения о том, что дерево пустое
    }
    инициализация указателя удаляемого узла –> deleteNode
    инициализация указателя ‘ребенка’ узла –> nodeChild

    if левый узел от node или правый узел от node пустой
    {
        deleteNode <– node;
    }
    else
    {
        deleteNode <– следующий минимальный узел от node
    }

    if левый от deleteNode не пуст
    {
        nodeChild <– левый от deleteNode
    }
    else
    {
        nodeChild <– правый от deleteNode
    }

    if nodeChild не пуст
    {
        родитель nodeChild <– родитель deleteNode
    }
```

```
    if родитель deleteNode пуст
    {
        вершина <- nodeChild;
    }
    else
    {
        if deleteNode равен левому от родителя deleteNode
        {
            левый от родителя deleteNode <- nodeChild;
        }
        else
        {
            правый от родителя deleteNode <- nodeChild;
        }
    }

    if deleteNode не равен node
    {
        слово из node <- слово из deleteNode;
    }
}
```

- **<u>Идентификатор доступа public:</u>**

1. Конструктор;
2. Деструктор;
3. Определение количества слов в словаре (тип – целочисленный параметр);
4. Вставка нового слова в словарь (тип – void; параметры – слово, список-переводов);
5. Поиск слова в словаре (тип – bool; параметр – слово);
6. Удаление слова из словаря (тип – void; параметр – слово);
7. Вывод дерева слов в консоль (тип – void; параметр – ссылка на стандартный поток вывода, указатель на узел вершины дерева);
8. Вывод перевода слова в консоль (тип – void; параметр – слово);

Деструктор

```
~BINARY_SEARCH_TREE()
{
    вызов функции рекурсивного освобождения памяти с
         параметром вершины дерева (DeleteSubtree(вершина))
}
```

Определение количества слов в словаре

```
GET_COUNT()
{
     вернуть функцию определения количества узлов с параметром
          вершины дерева (GET_COUNT_SUB_TREE(вершина))
}
```

Поиск слова в словаре

```
слово для поиска -> key
ссылка на список -> list

SEARCH(key, list)
{
     вернуть истину, если SEARCH_NODE(key) не равен nullptr
     иначе вернуть ложь
}
```

Удаление слова из словаря

```
слово для удаления -> key
ссылка на список -> list

DELETE_KEY(key)
{
     if слово найдено в словаре
     {
          инициализация указателя удаляемого узла с данными key ->
             toDelete
          удаление узла toDelete
     }
     else
     {
          вывод сообщения об отсутствии узла с таким словом в
             дереве
     }
}
```

Вывод дерева слов в консоль

```
ссылка на стандартный поток вывода <- out

PRINT(out)
{
     вывод информации хранящейся в дереве с вершины
}
```

Вставка нового слова в словарь

```
слово для вставки -> key
ссылка на список -> list

INSERT(key, list)
{
    инициализация указателя на головной(текущий) узел -> current
    инициализация указателя на узел с данными для вставки (key,
        list) -> insert
    инициализация указателя на родительский узел -> parent <-
        nullptr

    while current не пуст
    {
        присвоить parent <- current
        if key меньше слова в current
        {
            переместить current влево
        }
        else
        {
            переместить current вправо
        }
    }

    присвоить родительскому узлу insert <- parent

    if parent пуст
    {
        присвоить вершине дерева <- insert;
    }
    else
    {
        if key меньше слова в parent
        {
            присвоить левому узлу parent <- insert
        }
        else
        {
            присвоить левому узлу parent <- insert
        }
    }
}
```

Вывод перевода слова в консоль

```
слово для вывода перевода <- key

PRINT_NODE_TRANSLATION(key)
{
     инициализация указателя на узел с данными key -> node
     вывод списка из узла node
}
```

Примеры операций со словарем с использованием методов класса List:

- Вывод переводов:

```
Word translation:
1, 2, 3, 4, 5, 6
```

- Добавление перевода в список с сортировкой

```
List before adding new word:
1, 2, 3, 4, 6

List after adding new word:
1, 2, 3, 4, 5, 6
```

добавить <5>

- Определение количества переводов у слова

```
List:
1, 2, 3, 4, 5, 6

List size:
6
```

Примеры операций со словарем с использованием методов класса BinarySearchTree:

- Поиск слова в словаре:

```
Word:
d

Dictionary:
(e(d(b(a()())(c()()))())(h()()))

Serch result:
yes
```

- Вывод словаря слов:

```
Word translation:
(e(d(b(a()())(c()()))())(h()()))
```

- Определение количества слов в словаре

```
Dictionary:
(e(d(b(a()())(c()()))())(h()()))

Size:
6
```

- Добавление нового слова в словарь

```
Dictionary before adding word:
(e(d(b(a()())(c()()))())(h()()))

Dictionary after adding word:
(e(d(b(a()())(c()()))())(h(g()())()))
```

добавить “g”

- Удаление слова из словаря

```
Dictionary before deleting word:
(e(d(b(a()())(c()()))())(h()()))

Dictionary after deleting word:
(h(d(b(a()())(c()()))()))
```

удалить “e”