Лекция №3

Кросс-платформенное программирование

Двухсвязный список

Многие проблемы при работе с односвязным списком вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная идея – хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Для доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списка (**Head**) и на «хвост» - последний элемент (**Tail**).

Двухсвязный список

Общий принцип организации двухсвязного списка

Двухсвязный список

Наличие двух ссылок вместо одной предоставляет несколько преимуществ. Наиболее важное из них состоит в том, что перемещение по списку возможно в обоих направлениях. Это упрощает работу со списком, в частности, вставку и удаление. Помимо этого, пользователь может просматривать список в любом направлении.

Еще одно преимущество имеет значение только при некоторых сбоях. Поскольку весь список можно пройти не только по прямым, но и по обратным ссылкам, то в случае, если какая-то из ссылок станет неверной, целостность списка можно восстановить по другой ссылке.

Двухсвязный список

Каждый узел содержит (кроме полезных данных) также ссылку на следующий за ним узел (поле next) и предыдущий (поле prev). Поле next у последнего элемента и поле prev у первого содержат NULL. Узел объявляется так:

```
struct Node {
int data; // область данных
Node *next, *prev;
// ссылки на соседние узлы
};
```

Добавление узла в начало списка

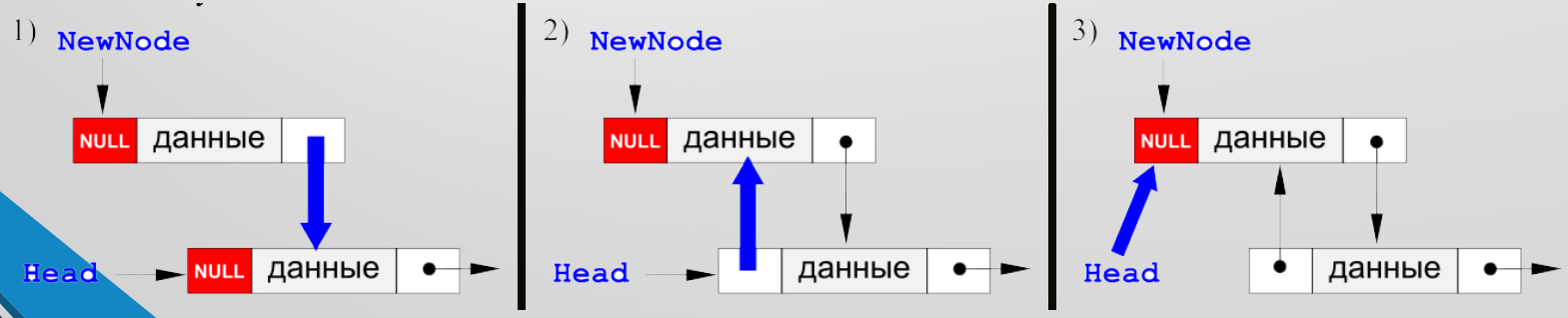
При добавлении нового узла NewNode в начало списка необходимо

1) установить ссылку next узла NewNode на голову существующего списка и его ссылку prev в NULL;

2) установить ссылку prev бывшего первого узла (если он существовал) на NewNode;

3) установить голову списка на новый узел;

4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.



Добавление узла в начало списка

```
void AddFirst(Node* NewNode)
{
   NewNode->next = Head;
   NewNode->prev = NULL;
   if ( Head ) Head->prev = NewNode;
   Head = NewNode;
   if ( ! Tail ) Tail = Head;
   // этот элемент – первый
}
```

Добавление узла в конец списка

Благодаря симметрии добавление нового узла NewNode в конец списка проходит совершенно аналогично, в процедуре следует везде заменить Head на Tail и наоборот, а также поменять prev и next.

Добавление узла в произвольное место списка

1) установить ссылки нового узла на следующий за данным (next) и предшествующий ему (prev);

2) установить ссылки соседних узлов так, чтобы включить NewNode в список.

Поиск узла в списке

Проход по двусвязному списку может выполняться в двух направлениях – от головы к хвосту (как для односвязного) или от хвоста к голове.

Для движения по списку часто испольуется следующая конструкция

```
Node *temp = Head;
while (temp != NULL)
{
// Здесь выполняются какие-либо действия с
// текущим (temp) элементом списка
temp = temp->next; // Шаг вперед по списку
}
```

Удаление узла

Эта процедура также требует ссылки на голову и хвост списка, поскольку они могут измениться при удалении крайнего элемента списка. На первом этапе устанавливаются ссылки соседних узлов (если они есть) так, как если бы удаляемого узла не было бы. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке внизу. Отдельно проверяется, не является ли удаляемый узел первым или последним узлом списка.

Удаление узла

```
void Delete(Node* OldNode)
{
    if (Head == OldNode)
    {   Head = OldNode->next; // удаляем первый элемент
        if ( Head )
            Head->prev = NULL;
        else Tail = NULL; // удалили единственный элемент
    }
    else
    {   OldNode->prev->next = OldNode->next;
        if ( OldNode->next )
            OldNode->next->prev = OldNode->prev;
        else Tail = NULL; // удалили последний элемент
    }
    delete OldNode;
}
```

Циклические списки

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель next последнего элемента указывает на первый элемент, и (для двусвязных списков) указатель prev первого элемента указывает на последний. В таких списках понятие «хвоста» списка не имеет смысла, для работы с ним надо использовать указатель на «голову», причем «головой» можно считать любой элемент.

Бинарные деревья

Бинарное (двоичное) дерево (binary tree) — это упорядоченное дерево, каждая вершина которого имеет не более двух поддеревьев, причем для каждого узла выполняется правило: в левом поддереве содержатся только ключи, имеющие значения, меньшие, чем значение данного узла, а в правом поддереве содержатся только ключи, имеющие значения, большие, чем значение данного узла.

Бинарное дерево является рекурсивной структурой, поскольку каждое его поддерево само является бинарным деревом и, следовательно, каждый его узел в свою очередь является корнем дерева.

Бинарное дерево

Основные понятия

Предком для узла x называется узел дерева, из которого существует путь в узел x.

Потомком узла x называется узел дерева, в который существует путь (по стрелкам) из узла x.

Родителем для узла x называется узел дерева, из которого существует непосредственная дуга в узел x.

Сыном узла x называется узел дерева, в который существует непосредственная дуга из узла x.

Уровнем узла x называется длина пути (количество дуг) от корня к данному узлу. Считается, что корень находится на уровне 0.

Листом дерева называется узел, не имеющий потомков.

Внутренней вершиной называется узел, имеющий потомков.

Высотой дерева называется максимальный уровень листа дерева.

Упорядоченным деревом называется дерево, все вершины которого упорядочены (то есть имеет значение последовательность перечисления потомков каждого узла).

Описание вершины

Вершина дерева, как и узел любой динамической структуры, имеет две группы данных:

полезную информацию и ссылки на узлы, связанные с ним. Для двоичного дерева таких ссылок будет две – ссылки на левого и правого потомка. В результате получаем структуру, описывающую вершину (предполагая, что полезными данными для каждой вершины является одно целое число):

```
struct Node
{
   int data;
   Node *left, *right;
};
```

Обход дерева

Одной из необходимых операций при работе с деревьями является обход дерева, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют три варианта обхода:

1) КЛП (корень – левое – правое): сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем – правое;

2) ЛКП (левое – корень – правое): сначала посещается левое поддерево, затем корень, а затем – правое;

3) ЛПК (левое – правое – корень): сначала посещается левое поддерево, затем правое, а затем – корень.

Для примера ниже дана рекурсивная процедура просмотра дерева в порядке ЛКП. Обратите внимание, что поскольку дерево является рекурсивной структурой данных, при работе с ним естественно широко применять рекурсию.

Обход дерева

```
void PrintLKP(Node* Tree)
{
if ( ! Tree ) return;
// пустое дерево – окончание рекурсии
PrintLKP(Tree->left);
// обход левого поддерева
printf("%d ", Tree->key);
// вывод информации о корне
PrintLKP(Tree->right);
// обход правого поддерева
}
```

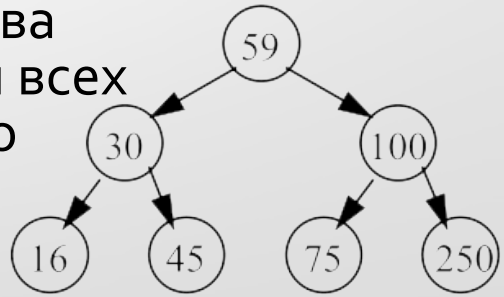
Поиск с помощью дерева

Деревья очень удобны для поиска в них информации. Однако для быстрого поиска требуется предварительная подготовка – дерево надо построить специальным образом.

Пусть есть массив: [59, 100, 75, 30, 16, 45, 250].

Теперь предположим, что данные организованы в виде дерева, показанного на рисунке. Такое дерево (оно называется дерево поиска) обладает следующим важным свойством:

Значения ключей всех вершин левого поддерева вершины x меньше ключа x, а значения ключей всех вершин правого поддерева x больше или равно ключу вершины x.



Для поиска нужного элемента в таком дереве требуется не более 3 сравнений вместо 7 при поиске в списке или массиве, то есть поиск проходит значительно быстрее. С ростом количества элементов эффективность поиска по дереву растет.

Алгоритм построения бинарного дерева

1. Сравнить ключ очередного элемента массива с ключом корня.
2. Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен, то в правое.
3. Если текущее дерево пустое, создать новую вершину и включить в дерево.

Реализация алгоритма

```
void AddElem(Elem** node, Elem *data)
{
    // Если лист -- добавляем элемент
    if(*node == NULL)
        *node = data;
    else
    {
        // В какую ветвь?
        if ((*node)->data > data->data)
            AddElem(&((*node)->left), data);
        else
            AddElem(&((*node)->right), data);
    }
}
```

Сортировка с помощью дерева поиска

Если дерево поиска построено, очень просто вывести отсортированные данные. Обход типа ЛКП (левое поддерево – корень – правое поддерево) даст ключи в порядке возрастания, а обход типа ПКЛ (правое поддерево – корень – левое поддерево) – в порядке убывания.

Максимум – это самый правый лист, минимум – самый левый.

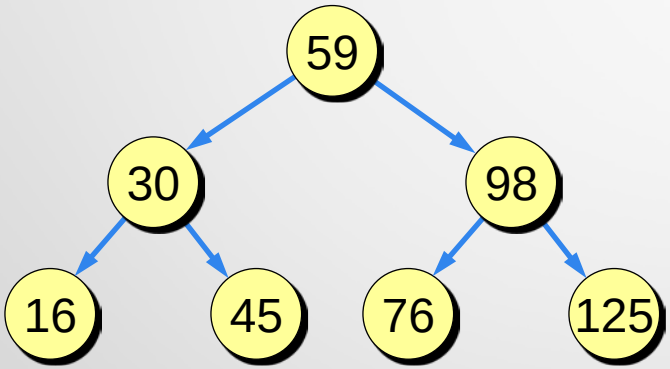
Двоичные деревья поиска

Поиск в массиве (N элементов):

При каждом сравнении отбрасывается 1 элемент.

Число сравнений – ***N***.

Поиск по дереву (N элементов):



При каждом сравнении отбрасывается половина оставшихся элементов.

Число сравнений ~ $\log_2 N$.

быстрый поиск

1) нужно заранее построить дерево;
2) желательно, чтобы дерево было минимальной высоты.

Поиск по дереву

```
Node* Search(Node* Tree, int x)
{
if (! Tree) return NULL;
if (x == Tree->data)
      return Tree;
if (x < Tree->data)
  return Search(Tree->left, x);
else
  return Search(Tree->right, x);
}
```

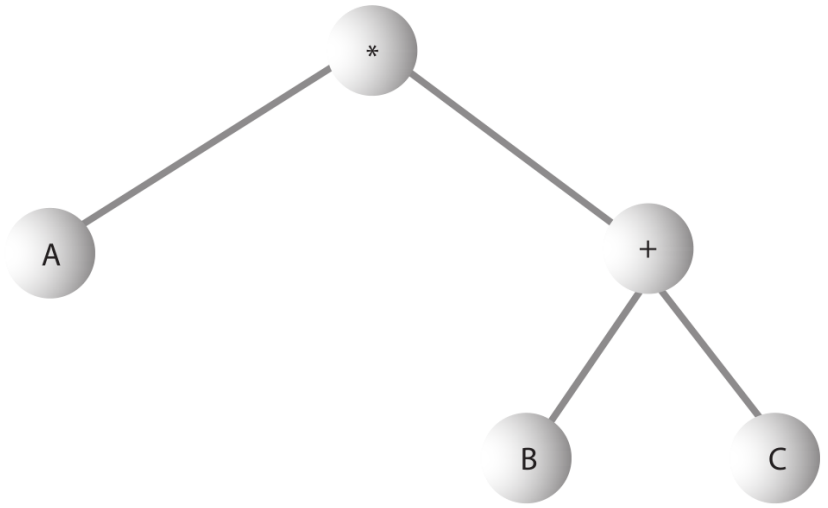
Временная сложность операций с бинарным деревом

Отлично	Хорошо	Удовлетворительно	Плохо	Ужасно

Структура данных	Временная сложность							
	Средняя				Худшая			
	Доступ	Поиск	Вставка	Удален.	Доступ	Поиск	Вставка	Удален.
Массив (Array)	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Стек (Stack)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Очередь (Queue)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Связанный список (Linked list)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Бинарное дерево (B-tree)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Использование бинарного дерева

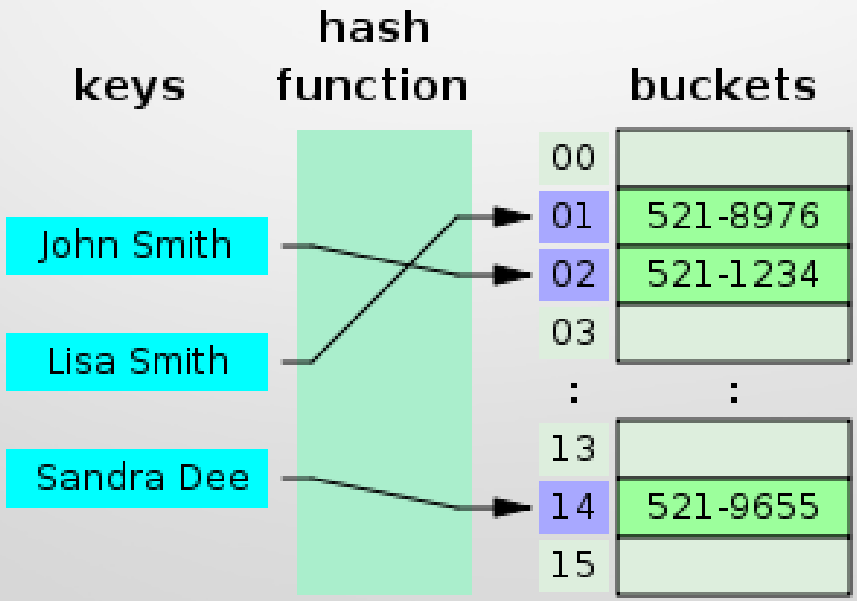
- Двоичное дерево (но не дерево двоичного поиска!) может использоваться для представления алгебраических выражений с бинарными операторами +, –, / и *.
- В корневом узле хранится оператор, а в других узлах — имя переменной (A, B или C) или другой оператор. Каждое поддерево представляет действительное алгебраическое выражение.



Инфиксная запись: A*(B+C)
Префиксная запись: *A+BC
Постфиксная запись: ABC+*

Хэш-таблицы

- Хеш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.



Хэш-таблицы

- Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время O(1). Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хеш-таблицы: увеличить значение размера массива и заново добавить в пустую хеш-таблицу все пары.

Временная сложность операций с хэш-таблицей

Отлично	Хорошо	Удовлетворительно	Плохо	Ужасно

Структура данных	Временная сложность							
	Средняя				Худшая			
	Доступ	Поиск	Вставка	Удален.	Доступ	Поиск	Вставка	Удален.
Массив (Array)	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Стек (Stack)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Очередь (Queue)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Связанный список (Linked list)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Бинарное дерево (B-tree)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Хэш-таблица (hash table)	–	O(1)	O(1)	O(1)	–	O(n)	O(n)	O(n)

Сложность различных алгоритмов для сортировки массивов

Отлично	Хорошо	Удовлетворительно	Плохо	Ужасно

Алгоритм сортировки	Временная сложность (Время выполнения)			Пространственная сложность (дополнительная память)
	Лучшая	Средняя	Худшая	Худшая
Пузырек (Bubble Sort)	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Простая вставка (Insertion Sort)	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Метод Шелла (Shell Sort)	O(n)	$O(n\log^2(n))$	$O(n\log^2(n))$	O(1)
Слиянием (Mergesort)	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Быстрая (Quicksort)	O(n log(n))	O(n log(n))	$O(n^2)$	O(log(n))
????? (где же ты?)	O(n log(n))	O(n log(n))	O(n log(n))	O(1)