Алгоритмы и структуры данных

Лекция 7
Списки. Деревья.

Сергей Леонидович Бабичев

План лекции

1. Структура данных «список».
2. Списки с пропусками.
3. Деревья. Обход деревьев.

Структура данных «список».

Списки

Список — структура данных, которая реализует абстракции:

- `insertAfter` — добавление элемента за текущим.
- `insertBefore` — добавление элемента перед текущим.
- `insertToFront` — добавление элемента в начало списка.
- `insertToBack` — добавление элемента в конец списка
- `find` — поиск элемента
- `size` — определение количества элементов

Списки: реализация

Для реализации списков обычно требуется явное использование указателей.

```
struct linkedListNode {
  someType data;
  linkedListNode *next;
};
```

Внутренние операции создания элементов — через `malloc`, `calloc`, `new`.

```
...
linkedListNode *item = new linkedListNode();
item->data = myData;
...
```

Списки: представления

Различные варианты представлений:
В линейном виде:

В виде кольца:

Списки: сложность

Стоимость операций:

Операция	Время	Память
`insertAfter`	`O(1)`	`O(1)`
`insertBefore`	`O(N)`	`O(1)`
`insertToFront`	`O(1)`	`O(1)`
`insertToEnd`	`O(N)`	`O(1)`
`find`	`O(N)`	`O(1)`
`size`	`O(N)`	`O(1)`

Списки: создание

```
typedef double myData;

linkedListNode *list_createNode(myData data) {
  linkedListNode *ret = new linkedListNode();
  ret->data = data;
  ret->next = nullptr;
  return ret;
}
```

Создание списка из одного элемента:

```
linkedListNode *head = list_createNode(555.666);
```

Списки: добавление

Добавление элемента в хвост списка:

```
linkedListNode *oth = list_createNode(123.45);
```

```
head->next = oth;
```

Списки: добавление

Добавление элемента в хвост списка, состоящего из нескольких элементов:

Проход по элементам до нужного (`traversal`, `walk`):

```
linkedListNode *ptr = head;
while (ptr->next != nullptr) {
  ptr = ptr->next;
}
ptr->next = oth;
```

Списки: добавление

Заключительное состояние после вставки.

Сложность операции — $O(N)$

Списки: добавление

Вставка `insertAfter` *ЗА* конкретным элементом `p1` примитивна.

```
oth->next = p1->next;
p1->next = oth;
```

Списки: добавление

Вставка *ПЕРЕД* известным элементом p2 сложнее:

```
linkedListNode *ptr = head;
while (ptr->next != p2) {
  ptr = ptr->next;
}
oth->next = p2;
ptr->next = oth;
```

Списки: удаление

Удаление элемента p2 — непростая операция.

- Нужно найти удаляемый элемент и его предшественника:

```
linkedListNode *ptr = head;
while (ptr->next != p2) {
  ptr = ptr->next;
}
// ptr - prev to p2
```

Списки: удаление

Удаление элемента из списка.

- Переместить указатели.

```
ptr->next = p2->next;
delete p2;
```

Списки: размер

Операция `size` — две возможности:

- Через операцию `walk` до `NULL`:

```
linkedListNode *ptr = head;
int size = 0;
while (ptr != nullptr) {
  ptr = ptr->next;
  size++;
}
return size;
```

- Вести размер списка в структуре данных. Потребуется изменить все методы вставки/удаления.

Списки: альтернативное представление

- При нашем представлении требуется всегда различать, работаем ли мы с головой списка или с другим элементом. При смене головы списка приходится заменять все указатели в программе.
- Существуют различные способы представления списков.
- Для абстрактного типа данных удобнее иметь список с неизменной головой.

- Это — пустой список, содержащий ноль элементов.

Списки: альтернативное представление

- Список, состоящий из одного элемента `p1`.
- Такое представление упрощает реализацию за счёт одного дополнительного элемента.

Списки: сложность

Ещё раз оценим сложность основных операций:

- Вставка элемента в голову списка — $O(1)$
- Вставка элемента в хвост списка — *O(N)*
- Поиск элемента — $O(N)$
- Удаление известного элемента — *O(N)*
- Вставка элемента ЗА известным — $O(1)$
- Вставка элемента ПЕРЕД известным — *O(N)*

Можно ли улучшить худшие случаи?

Списки: двусвязные списки

Худшие случаи можно улучшить, если заметить, что операция «слева-направо» более эффективно реализуется, чем «справа-налево» и восстановить симметрию.

Списки: двусвязные списки: сложность

Для двусвязного списка сложность такая:

- Вставка элемента в голову списка — $O(1)$
- Вставка элемента в хвост списка — *O(1)*
- Поиск элемента — $O(N)$
- Удаление известного элемента — *O(1)*
- Вставка элемента ЗА известным — $O(1)$
- Вставка элемента ПЕРЕД известным — *O(1)*

Списки: двусвязные списки: вставка

Операции вставки и удаления усложняются:
Для вставки элемента `oth` после элемента `p1`:

1. Подготавливаем вставляемый элемент.
2. Сохраняем указатель `s = p1->next`
3. `oth->prev = p1`
4. `oth->next = s`
5. `s->prev = oth`
6. `p1->next = oth`

Списки: двусвязные списки: удаление

Для удаления элемента `p1` из списка:

1. Сохраняем указатель `s = p1->next`
2. `s->prev = p1->prev`
3. `p1->prev->next = s`
4. Освобождаем память элемента `p1`

Списки: использование

Когда используют списки? Когда нужно представлять быстро изменяющееся множество объектов.

- Пример из математического моделирования: множество машин при моделировании автодороги. Они:
  - появляются на дороге (вставка в начало списка)
  - покидают дорогу (удаление из конца списка)
  - перестраиваются с полосы на полосу (удаление из одного списка и вставка в другой)
- Пример из системного программирования: представление множества исполняющихся процессов, претендующих на процессор. Представление множества запросов ввода/вывода. Важная особенность: лёгкий одновременный доступ от множества процессоров.

Списки: использование: менеджер памяти

Одна из реализаций выделения/освобождения динамической памяти (`calloc/new/free/delete`).

- Вначале свободная память описывается пустым списком.
- Память в операционной системе выделяется *страницами*.
- При заказе памяти:
  - если есть достаточный свободный блок памяти, то он разбивается на два подблока, один из которые помечается занятым и возвращается в программу;
  - если нет достаточной свободной памяти, запрашивается несколько страниц у системы и создаётся новый элемент в конце списка (или изменяется старый).

На практике применяется несколько списков, в зависимости от размера заявки.

Связные списки как хранилище

- Сложность операций:
  - ***Create*** — $O(1)$
  - ***Read*** — $O(N)$
  - ***Update*** — $O(N)$
  - ***Delete*** — $O(N)$

Параллельное использование алгоритмов поиска. Списки с пропусками.

Параллельное использование

- При параллельном программировании к одному элементу данных может обратиться несколько потоков.
- Результат при этом может быть недетерминирован.

```
int a = 0, b = 0;

//thread 1
b = 2;
a = b + 1;

//thread 2
a = 4;
b = a - 3;
```

Параллельное использование

- Критерий Бернстайна: Поместим объекты, которые читаются в потоке i в множество R_i, а те, которые пишутся, в множество W_i.
- Для нашего кода $R_1 = \{b\}$, $W_1 = \{a, b\}$, $R_2 = \{a\}$, $W_2 = \{a, b\}$.
- Критерий гласит, что если все пересечения множеств $R_1 \cap W_2$, $R_2 \cap W_1$, $W_1 \cap W_2$ пусты, то конфликтов (*race conditions*) не возникнет.
- В нашем случае: $R_1 \cap W_2 = \{a\}$, $R_2 \cap W_1 = \{a\}$, $W_1 \cap W_2 = \{a, b\}$, то есть *race conditions* возможны.

Параллельное использование

- Одно из средство избежать *race conditions* — использование атомарных операций.
- Существуют машинные команды типа *Compare-And-Swap*, исполняющиеся атомарно.
- Они позволяют атомарно обменять две ячейки памяти, которые, возможно, содержат указатели.
- При вставке в односвязный список достаточно атомарных операций для замены цепочки указателей.
- Односвязный список — идеальная структура данных для параллельного программирования.

Параллельное использование

- Пока: операция поиска в односвязном списке $T(N) = O(N)$
- Пока: операции вставки и удаления в односвязном списке $T(N) = O(N)$
- Требуется: по возможности сохранить свойства операций вставки и удаления в лучшем случае и ускорить все операции.

Списки с пропусками

Рассмотрим следующую структуру данных:

- Это — несколько списков, организованных в виде списков.
- Каждый следующий список примерно в два раза короче предыдущего и он пропускает примерно половину элементов предыдущего.

Списки с пропусками

- Поиск существующего элемента.

Списки с пропусками

- Поиск несуществующего элемента.

Списки с пропусками

- Вставка элемента.

Списки с пропусками

- Удаление элемента. Поиск и пометка столбца.

Списки с пропусками

- Удаление элемента. Удаление из строк.

Списки с пропусками

- Удаление элемента. Заключительное удаление.

Списки с пропусками как хранилище

- Сложность операций:
  - ***Create*** — $O(\log N)$
  - ***Read*** — $O(\log N)$
  - ***Update*** — $O(\log N)$
  - ***Delete*** — $O(\log N)$

Структура данных «дерево».

Деревья: особенности

Основная особенность деревьев — наличие нескольких наследников.
По максимальному числу наследников деревья делятся на

- двоичные (бинарные)
- троичные (тернарные)
- N-ричные

```
struct tree {
  tree *children[3];
  myType data;
  ...
};
```

Деревья: соглашения

- Любое `N`-ричное дерево может представлять деревья меньшего порядка.
- Соглашение: если наследника нет, соответствующий указатель равен `NULL`.
- Деревья `1`-ричного порядка существуют (списки).

Деревья: троичное дерево

Пример дерева троичного дерева или дерева 3-порядка.

Деревья: классификация

- Условно все элементы дерева делят на две группы:
  - ***Вершины***, не содержащие связей с потомками.
  - ***Узлы***, содержащие связи с потомками.
- Второй вариант — все элементы дерева называют ***узлами***, а ***вершина*** — частный случай ***узла***, ***терминальный узел***.
- Ещё термины:
  - ***Родитель (parent)***
  - ***Дети (children)***
  - ***Братья (sibs)***
  - ***Глубина (depth)***

$$D_{node} = D_{parent} + 1$$

Деревья: создание узла

- Добавим метод создания элемента (узла) дерева:

```
struct tree {
  string data;
  tree *child[3]{nullptr,nullptr,nullptr}
  tree(string init) { //  constructor
    data = init;
  }
  ...
};
```

Деревья: пример построения

- Дерево на примере строится, например, так:

```
tree *root = new tree("foo");
root->child[0] = new tree("bar");
root->child[1] = new tree("abr");
root->child[2] = new tree("aca");
root->child[0]->child[0] = new tree("dab");
root->child[0]->child[1] = new tree("ra");
```

Деревья: пример использования

Использование деревьев:

- Для представления выражений в языках программирования.

Деревья: вариант представления

- Вариант хранения N-дерева: массив.
  - Все узлы нумеруются, начиная с 0.
  - Для узла с номером K номера детей

$$K \cdot N + 1 \ldots K \cdot N + N$$

  - Для 2-дерева корневой узел = 0, дети 1-го уровня ($Depth = 2$) 1 и 2 ...

Часто удобнее нумеровать с 1. Тогда дети нумеруются

$$[K \cdot N \ldots (K + 1) \cdot N)$$

Деревья: нумерация

Для дерева выражений нумерация будет такой:

Деревья: альтернативное представление

- Представление в виде массива (фрагмент):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...	22
=	x	-			*	sin					35	y	+								

- Количество памяти = $O(2^{D_{max}})$
- Невыгодно при разреженном дереве

Деревья: обход

- Алгоритмы работы с деревьями часто рекурсивны.
- Всего существует 6=3! способов обхода бинарного дерева.
- На практике применяют четыре основных варианта рекурсивного обхода:
  - Прямой
  - Симметричный
  - Обратный
  - Обратно симметричный

Деревья: обход

Бинарное дерево.

```
struct tree {
  string data;
  tree *left = nullptr;
  tree *right = nullptr;
  tree(string const &init) { //  Constructor
    data = init;
  }
};
```

Деревья: обход

Прямой способ обхода.

```
void walk(tree *t) {
  work(t);
  if (t->left != nullptr) walk(t->left);
  if (t->right != nullptr) walk(t->right);
}
```

Деревья: обход

Симметричный способ обхода.

```
void walk(tree *t) {
  if (t->left != nullptr) walk(t->left);
  work(t);
  if (t->right != nullptr) walk(t->right);
}
```

Деревья: обход

Обратный способ обхода.

```
void walk(tree *t) {
  if (t->left != nullptr) walk(t->left);
  if (t->right != nullptr) walk(t->right);
  work(t);
}
```

Деревья: обход

Функция обработки может быть параметром.

```
using walkFunction = void (*)(tree *); // C++11 syntax
void walk(tree *t, walkFunction wf) {
  if (t->left != nullptr) walk(t->left, wf);
  if (t->right != nullptr) walk(t->right, wf);
  wf(t);
}
void printData(tree *t) {
  printf("t[%p]='%s'\n", t, t->data.c_str());
}
int main() {
  tree *root = new tree("foo");
  root->left = new tree("bar");
  root->right = new tree("abr");
  root->left->left = new tree("aca");
  root->left->left->left = new tree("dab");
  root->left->right = new tree("ra");
  walk(root, printData);
}
```

Деревья: обход

```
t[0x7ff0e1c03290]='dab'
t[0x7ff0e1c03260]='aca'
t[0x7ff0e1c032d0]='ra'
t[0x7ff0e1c03200]='bar'
t[0x7ff0e1c03230]='abr'
t[0x7ff0e1c031d0]='foo'
```

Деревья: обход

Вывод генеалогического дерева (обратно симметричный обход):

```
using walkFunction = void (*)(tree *t, int lev); // C++11 syntax

void walk(tree *t, walkFunction wf, int lev) {
  if (t->right != nullptr) walk(t->right, wf, lev+1);
  wf(t, lev);
  if (t->left != nullptr) walk(t->left, wf, lev+1);
}

void printData(tree *t, int lev) {
  for (int i = 0; i < lev; i++) {
    printf("  ");
  }
  printf("%s\n", t->data.c_str());
}
int main() {
  ...
  walk(root, printData, 0);
}
```

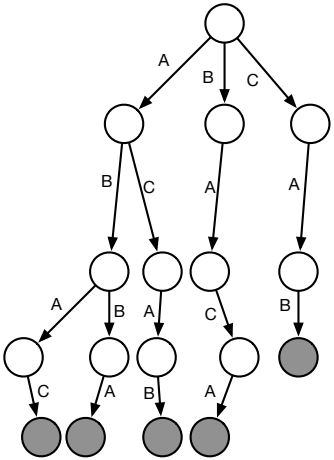
Деревья: обход

Вывод программы:

```
  abr
foo
    ra
  bar
    aca
      dab
```

Деревья: обход

- При использовании динамических структур данных для некоторых операций важно выбрать верный обход дерева.
- Вспомним префиксное дерево из второй лекции:



Деревья: обход

- Заказ памяти под поддеревья происходил динамически.
- Имелся узел, от которого шло построение дерева.
- Так как в данном дереве не хранится информация о том, кто является предком узла, корневой узел — центр всего построения.
- При операции освобождения памяти узла исказятся значения подузлов.

Деревья: динамическая память

- Напомним порядок выделения и уничтожения памяти в конструкторе и деструкторе:

```
struct node {
  node *children[3];
  bool  is_leaf;
  node();
  ~node();
};
```

Деревья: динамическая память

- Конструктор:

```
node::node() {
  children[0] = children[1] = children[2] = nullptr;
  is_leaf = false;
}
```

1. Система выделяет память из *кучи*, достаточную для хранения всех полей структуры.
2. После этого исполняется инициализация полей (написанный нами код).

Деревья: динамическая память

- Деструктор:

```
node::~node() {
  printf("node destructor is called\n");
}
```

1. Исполняется написанный нами код.
2. Система освобождает занятую память.
3. Обращение к освобождённой памяти приводит к ошибкам.

Деревья: динамическая память

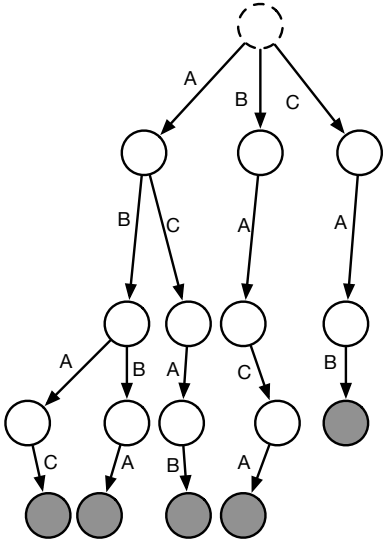
Деревья: динамическая память

- Удаление корневого узла приводит к тому, что остальные узлы останутся недоступны.
- Такие недоступные узлы называются *висячими ссылками* (*dangling pointers*).
- Ситуация, возникшая в программе называется *утечкой памяти* (*memory leak*).
- Чтобы не было утечки памяти, удаление узлов нужно производить с терминальных.

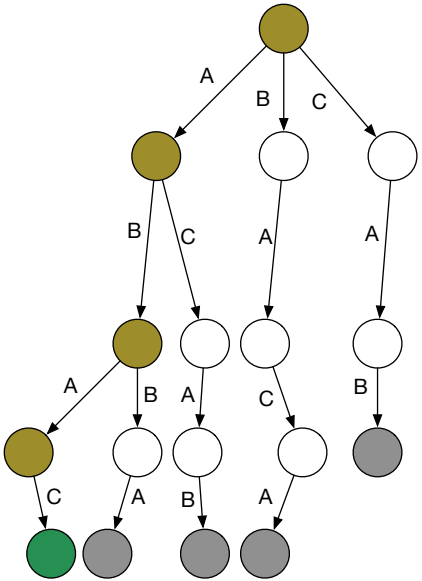
Деревья: освобождение памяти

```
void destroy(node *n) {
  for (int i = 0; i < 3; i++) {
    if (n->children[i] != nullptr) {
      destroy(n->children[i]);
    }
  }
  delete n;
}
```

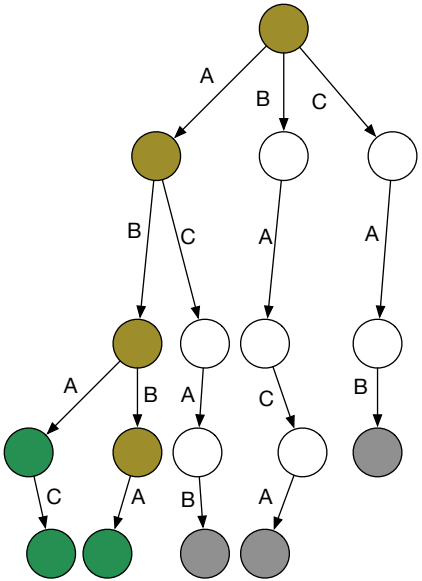
Деревья: освобождение памяти



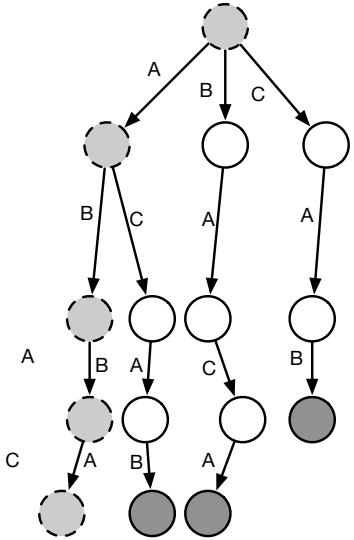
Деревья: освобождение памяти



Деревья: освобождение памяти



Деревья: освобождение памяти



Деревья: свойства

Свойства деревьев:

- Позволяют использовать быстро изменяющиеся структуры данных.
- Есть надежда, что операции вставки и удаления окажутся быстрыми (быстрее $O(N)$).
- Есть надежда, что операции поиска окажутся быстрыми (быстрее $O(N)$).

Спасибо за внимание.

Следующая лекция — сбалансированные деревья.